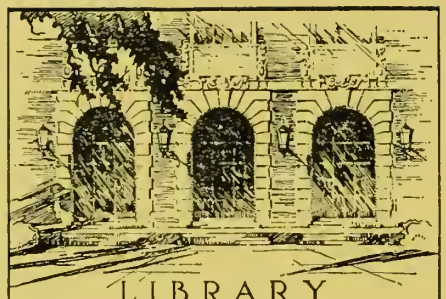
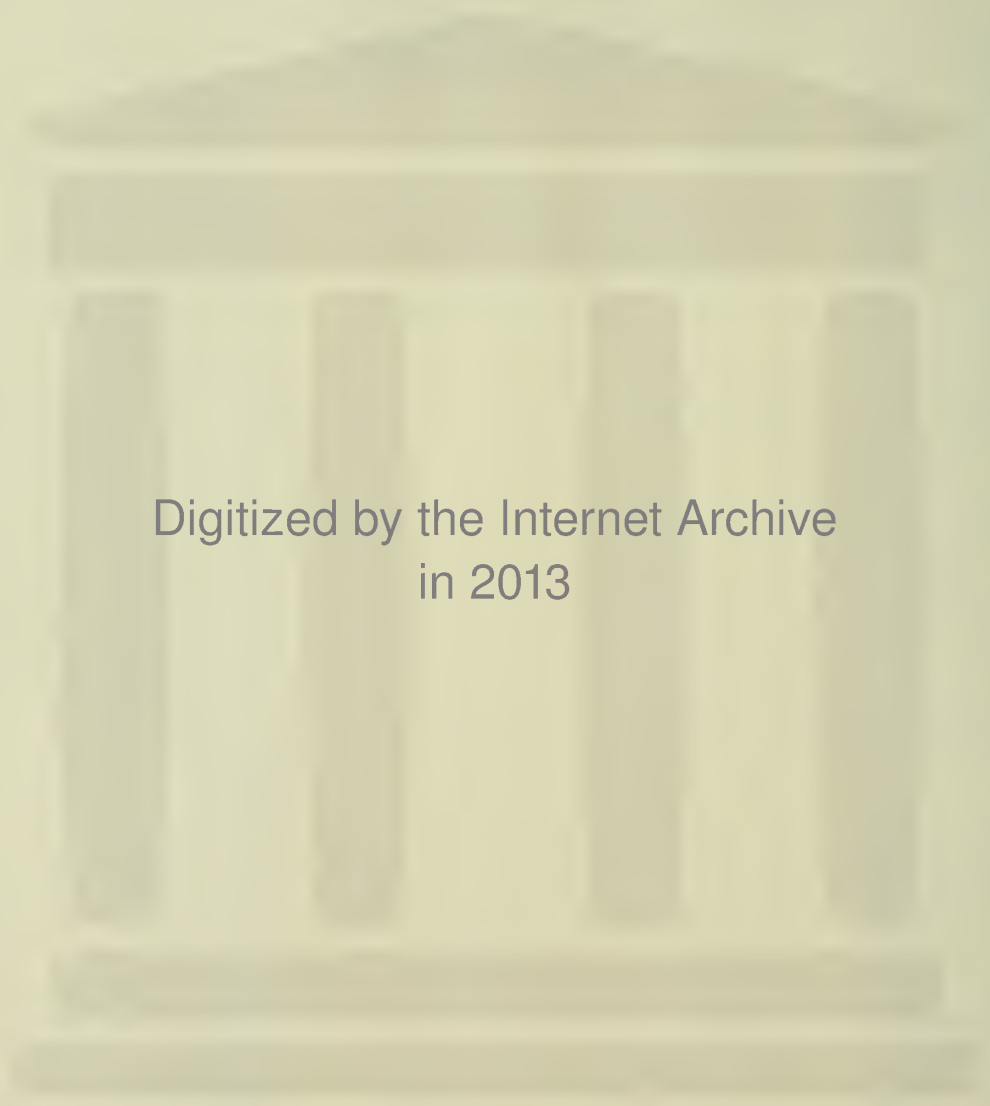




Report No. 258

ILLIAC IV Doc. No. 182

EIGEN-VALUE PROBLEMS

by

Ahmed Sameh

Luke Han

April 4, 1968

DEPARTMENT OF COMPUTER SCIENCE · UNIVERSITY OF ILLINOIS · URBANA, ILLINOIS

Report No. 258

ILLIAC IV Doc. No. 182

EIGEN-VALUE PROBLEMS

by

Ahmed Sameh

Luke Han

April 4, 1968

Department of Computer Science
University of Illinois
Urbana, Illinois 61801

(This work was supported in part by the Department of Computer Science, University of Illinois, Urbana, Illinois, and in part by the Advanced Research Projects Agency as administered by the Rome Air Development Center under Contract No. US AF 30(602)4144.)

ABSTRACT

The most used methods for solving algebraic eigen-value problems are discussed in detail in this report. These methods are Jacobi's, Householder's and QR-algorithms. Attempts are made to adapt or even modify the algorithm to take advantage of parallel computations.

A. Jacobi's Method

It proved to be one of the most effective methods on a parallel computer. However, it is limited to symmetric matrices with dominant principal diagonals. Evaluation of the eigen-values and eigen-vector is rather straightforward once the method converges.

B. Householder's Method

This method reduces the symmetric matrix to the tridiagonal form by means of elementary Hermitian orthogonal matrices. Once the matrix is in the tridiagonal form, evaluation of the eigen-values and eigen-vectors is performed by computational techniques that make almost full use of parallelism.

C. QR-Algorithm

The QR-algorithm is used for finding the eigen-values of unsymmetric matrices. For best results, the matrix is first reduced to an upper Hessenberg form, using Householder's method. Convergence is accelerated by performing a single origin shift; however, when the matrix has some complex conjugate eigen-values, a double origin shift is used in order to avoid complex conjugate shifts.

Flow charts and Tranquil language programs for all algorithms are included in the report.

TABLE OF CONTENTS

1. Jacobi's Method

1.1 Introduction

There are many methods for finding eigenvalues and eigenvectors of a symmetric matrix. Two of the most popular methods are Jacobi's method and Householder-Givens method. In order to apply them on a parallel computer effectively, we have made several modifications to both of the methods. In this section, we will discuss Jacobi's method.

1.2 Modification to the Classical Jacobi's Method

Instead of using the Classical Jacobi's Method, i.e., going through all the trouble to search for a pair (since it is symmetric) of largest off diagonal element and to eliminate them, we modified the method by eliminating all the off-diagonal elements of each 2 x 2 submatrices along the diagonal through orthogonal transformation. At each iteration, this modification will remove n elements of a n x n matrix. We eliminate the elements by determining an orthogonal matrix φ such that $\varphi A^i \varphi^t = A^{i+1}$ where A^{i+1} having zeros in the appropriate locations. To eliminate those elements, we choose

$$\varphi = \text{Diag.}\ (T_1, T_2, \ldots, T_{n/2})$$

where $$T_k = \begin{bmatrix} \cos \alpha_k & \sin \alpha_k \\ \sin \alpha_k & -\cos \alpha_k \end{bmatrix} \qquad K = 1, 2, \ldots, n/2$$

So $T = T^t$, $\varphi = \varphi^t$ and $TT^t = I$.

If matrix A^i is partitioned into 2 x 2 blocks as follows:

$$A^i = \begin{bmatrix} A_{11} & A_{12} & A_{13} & \cdots & A_{1\,n/2} \\ A_{12}^t & A_{22} & A_{23} & \cdots & A_{2\,n/2} \\ A_{13}^t & A_{23}^t & A_{33} & \cdots & A_{3\,n/2} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ A_{1\,n/2}^t & A_{2\,n/2}^t & A_{3\,n/2}^t & \cdots & A_{n/2\,n/2} \end{bmatrix}$$

Then $A^{i+1} = \varphi A^i \varphi^t$ will be

$$A^{i+1} = \begin{bmatrix} T_1 A_{11} T_1^t & T_1 A_{12} T_2^t & \cdots & T_1 A_{1n/2} T_{n/2}^t \\ T_2 A_{12}^t T_1^t & T_2 A_{22} T_2^t & \cdots & T_2 A_{2n/2} T_{n/2}^t \\ \vdots & \vdots & \cdots & \vdots \\ T_{n/2} A_{1n/2}^t T_1^t & T_{n/2} A_{2n/2}^t T_2^t & \cdots & T_{n/2} A_{n/2n/2} T_{n/2}^t \end{bmatrix}$$

Considering the diagonal submatrices $A_{kk}^{i+1} = T_k A_{kk}^i T_k^t$

$$= \begin{bmatrix} \cos\alpha_k & \sin\alpha_k \\ \sin\alpha_k & -\cos\alpha_k \end{bmatrix} \begin{bmatrix} a_{2k-1,2k-1} & a_{2k-1,2k} \\ a_{2k-1,2k} & a_{2k,2k} \end{bmatrix} \begin{bmatrix} \cos\alpha_k & \sin\alpha_k \\ \sin\alpha_k & -\cos\alpha_k \end{bmatrix}$$

$$= \begin{bmatrix} b_{2k-1,2k-1} & b_{2k-1,2k} \\ b_{2k-1,2k} & b_{2k,2k} \end{bmatrix}$$

where $b_{2k-1,2k-1} = a_{2k-1,2k-1} \cos^2\alpha_k + a_{2k-1,2k} \sin^2\alpha_k$

$$+ a_{2k,2k} \sin^2\alpha_k,$$

$$b_{2k,2k} = a_{2k-1,2k-1} \sin^2 \alpha_k - a_{2k-1,2k,} \sin^2 \alpha_k + a_{2k,2k} \cos^2 \alpha_k$$

and $$b_{2k-1,2k} = \frac{1}{2} \left(a_{2k-1,2k-1} - a_{2k,2k}\right) \sin^2 \alpha_k - a_{2k-1,2k} \cos^2 \alpha_k$$

Now to eliminate the off-diagonal terms, i.e., $b_{2k-1,2k} = 0$

$$\tan^2 \alpha_k = (a_{2k-1,2k})/\frac{1}{2} (a_{2k-1,2k-1} - a_{2k,2k})$$

$$\alpha_k = \frac{1}{2} \tan^{-1} \left(\frac{2 \alpha_{2k-1,2k}}{a_{2k-1,2k-1} - a_{2k,2k}}\right)$$

If we define that

$$Z_k = \frac{1}{2} \sqrt{1/([4a^2_{2k-1,2k}/(a_{2k,2k} - a_{2k-1,2k-1})^2] + 1)}$$

Then the rotation angles will be

$$\sin \alpha_k = \sqrt{\frac{1}{2} + Z_k} \quad ; \text{ and } \cos \alpha_k = \sqrt{\frac{1}{2} - Z_k}$$

and $$b_{\substack{2k-1,2k-1 \\ 2k\ ,2k}} = \frac{1}{2} (a_{2k-1,2k-1} + a_{2k,2k}) \pm \sqrt{a^2_{2k-1,2k} + \left(\frac{a_{2k-k,2k-1} - a_{2k,2k}}{2}\right)^2}$$

we also notice that

$$\text{tr } (A^i) = \text{tr } (A^{i+1})$$

and since the norm is invariant under an orthogonal transformation of the matrix. Therefore,

$$N^2 (A^{i+1}) = N^2 (A^i)$$

Since $$N^2 (A^i) = \text{tr} (\bar{A}^{i^t} A^i)$$

where $\bar{A}$ is the absolute of A. Also

$$N^2 (A_{kk}^{i+1}) = N^2(A_{kk}^i)$$

therefore, $b_{2k-1,2k-1}^2 + b_{2k,2k}^2 = a_{2k-1,2k-1}^2 + a_{2k,2k}^2 + 2\, a_{2k-1,2k}^2$

Hence by one orthogonal transformation, the sum of squares of the diagonal coefficients increases by the sum of squares of the (n) vanished terms.

Now, after one orthogonal transformation, the matrix will look as follows:

$$A^{i+1} = \left[\begin{array}{cc|cc|c|c} b_{11} & 0 & b_{13} & b_{14} & \cdots & b_{1,n-1}\, b_{1,n} \\ 0 & b_{22} & b_{23} & b_{24} & \cdots & b_{2,n-1}\, b_{2,n} \\ \hline b_{13} & b_{23} & b_{33} & 0 & \cdots & \\ b_{14} & b_{24} & 0 & b_{44} & \cdots & \\ \hline \multicolumn{4}{c|}{\cdots} & \cdots & \cdots \\ \hline \multicolumn{4}{c|}{\cdots} & b_{n-1,n-1} & 0 \\ \multicolumn{4}{c|}{\cdots} & 0 & b_{n,n} \end{array}\right]$$

Therefore, by shifting the second row to the bottom and the second column to the far right, we bring to the diagonal new submatrices with non-zero off-diagonal elements, and hence the matrix is ready for the second transformation. This shifting is repeated for n - 1 transformations. Then, we will shift

the first row to the bottom and first column to the far right and repeat the above mentioned shuffling process for m transformations, (the diagonal terms are sufficiently dominating). Therefore, the diagonal elements will be equal to the eigen-values, since the eigen-values are not changed by orthorgonal transformations. For comparing number of transformations required by this revised method with the classical Jacobi's method, see Appendix F, first Q.P.R. The eigen-vectors will be the column vectors of B, where

$$B = \varphi_1^t \varphi_2^t \cdots \varphi_m^t$$

because if we let the final matrix be F, then

$$\varphi_m \cdots \varphi_3 \varphi_2 \varphi_1 A \varphi_1^t \varphi_2^t \cdots \varphi_m^t = F$$

but $AX = \lambda X$

Since F is almost diagonal with λ's being its diagonal elements,

$\therefore$ $AX = FX$ where X is the matrix with eigen-vectors as its columns

Hence $A \varphi_1^t \varphi_2^t \cdots \varphi_m^t = F \varphi_1^t \varphi_2^t \cdots \varphi_m^t$

and $B = \varphi_1^t \varphi_2^t \cdots \varphi_m^t$ is the matrix with the corresponding eigen-vectors.

Thus, in practice one can follow the following four steps:

(i) Calculate orthogonal transformation matrix φ by finding n rotation angles and construct 2 X 2 submatrices on the diagonal and zeroes everywhere else.

(ii) Pre and post-multiply matrix A^i at i-th step by orthogonal transformation matrix φ^i to get A^{i+1}. At the same time, get B^{i+1} by $B^i \varphi^i$ where B^1, initial B, is identity.

(iii) Testing whether the ratio between the sum of the squares of the diagonal elements and the sum of the squares of the off-diagonal elements is greater than $(1/\zeta)$, where ζ is a given small number (.001 - .01). If greater, print out the eigen-values and the eigen-vectors from the present matrices A^m and B^m respectively, otherwise go to next step.

(iv) Shuffle the second row and second column to start another transformation. After (n-1) such shuffling, shuffle the first row and first column and start all over again. If after (n-1) shuffling of the first row and column, the diagonal elements are still not sufficiently dominant, the method fails.

Jacobi's method has proved to be suitable for diagonally-dominant matrices. Therefore, although Householder's method works for any symmetric matrix, due to the simplicity of Jacobi's method, it will be used for finding eigen-values and eigen-vectors for diagonally-dominant matrices.

1.3 High-Level Language Code and Flow Chart

For flow chart and high-level language code see ILLIAC IV Document Number 165.

2. Householder's Method

2.1 Introduction

This method is one of the most effective methods that exist for reducing a symmetric matrix to tri-diagonal form using elementary Hermitian orthogonal matrices. For a matrix of order n, there are (n-2) steps in this reduction, in the r^{th} of which [n-(r+1)] zeroes are introduced in the r^{th} row and r^{th} column without destroying the zeroes introduced in the previous steps.

2.2 Theoretical Background

The present problem is that, for a given vector x of order n, it is required to determine an elementary Hermitian matrix P such that pre-multiplication by P leaves the first component of x and eliminates the rest, i.e.,

$$Px = ke_1 \tag{2.1}$$

where

k = constant

$e_1 = \{1, 0, 0, \ldots, 0\}$ (column vector)

Let $$P = I - 2ww^t \tag{2.2}$$

where w is a column vector of order n and $w^t w = 1.0$. P is not only elementary Hermitian but also orthogonal for

$$\begin{aligned} P^t P &= (I - 2ww^t)^t (I - 2ww^t) \\ &= I - 4ww^t + 4w(w^t w)w^t \\ &= I \end{aligned}$$

from Equ. (2.1), it can be proved that the Euclidean norm of x is invariant, i.e.

$$S^2 = x_1^2 + x_2^2 + \ldots x_n^2 = k^2,$$

2-Norm of $Px = (Px)^t (Px) = (Px)^t (Px)$

$$= x^t P^t P_x = x^t P^t Px$$

$$= x^t x \quad \text{which is the 2-norm of } x,$$

Therefore, $k = \pm S$ (2.3)

hence, equation (2.1) yields,

$$\begin{aligned} x_1 - 2w_1 K &= \pm S \\ x_i - 2w_i K &= 0 \quad (i = 2,3,\ldots,n) \end{aligned} \tag{2.4}$$

where $K = w^t x$ which is a scalar. Squaring and adding equations (2.4), the value of the scalar K can be obtained

$$K = \sqrt{\frac{s^2 \pm x_1 s}{2}} \tag{2.5}$$

If $u^t = (x_1 \mp s, x_2, x_3, \ldots, x_n)$, equations (2.4) yield,

$$w = \frac{u}{2K} \tag{2.6}$$

and

$$P = I - \frac{uu^t}{2K^2} \tag{2.7}$$

For a matrix A of order n, suppose that the configuration of the matrix just before the rth step is as follows:

$$A_{r-1} = \left[\begin{array}{c|c} C_{r-1} & \begin{matrix} 0 \\ \hline b^t_{r-1} \end{matrix} \\ \hline \begin{matrix} 0 & b_{r-1} \end{matrix} & B_{r-1} \end{array}\right] \tag{2.8}$$

where C_{r-1} is a tri-diagonal matrix of size r, b_{r-1} is a column vector of order (n-r) and B_{r-1} is a matrix of size (n-r). The matrix P_r of the rth transformation may be expressed in the form,

$$P_r = \left[\begin{array}{c|c} I & 0 \\ \hline 0 & Q_r \end{array}\right] = \left[\begin{array}{c|c} I & 0 \\ \hline 0 & I-2v_r v_r^t \end{array}\right] \tag{2.9}$$

where v_r is a unit vector having (n-r) components, $w_r^t = (0 \mid v_r^t)$. Hence, we have

$$A_{rs} = P_r A_{r-1} P_r^t = \left[\begin{array}{c|c} C_{r-1} & \begin{matrix} 0 \\ \hline C_r^t \end{matrix} \\ \hline \begin{matrix} 0 & C_r \end{matrix} & Q_r B_{r-1} Q_r \end{array}\right] \tag{2.10}$$

where $c_r = Q_r\, b_{r-1}$

If we chosse v_r such that all components of c_r vanish except for its first component, then A_r is a tri-diagonal matrix as far as its first (r+1) rows and columns are concerned.

Making use of equations (2.2) through (2.7), the method can be organized in a fast systematic manner,

$$P_r = I - (u_r\, u_r^t / 2K_r^2)$$

where

$$\begin{aligned} u_{ir} &= 0 \qquad (i = 1, 2, \ldots, r) \\ u_{r,r+1} &= a_{r,r+1} \mp S_r \\ u_{ir} &= a_{ri} \qquad (i = r+2, \ldots, n) \end{aligned} \tag{2.11}$$

where

$$S_r = \left(\sum_{i=r+1}^{n} a_{ri}^2 \right)^{\frac{1}{2}}$$

$$K_r = \left[(S_r^2 \mp a_{r,r+1}\, S_r)/2 \right]^{\frac{1}{2}}$$

In equations (2.11) the sign associated with S_r is to be taken as that of $a_{r,r+1}$.

The method discussed above involves quite a large number of multiplications which may be necessary in case of reducing a non-symmetric matrix to an upper Hessenberg form. However, for symmetric matrices, the above method can be modified slightly in order to take advantage of the present symmetry.

$$A_r = P_r\, A_{r-1}\, P_r \qquad (P_r^t = P_r)$$

using equation (2.7), and denoting that,

$$(A_{r-1}\, u_r)/2K_r^2 = p_r \quad \text{and} \quad (u_r^t\, A_{r-1})/2K_r^2 = p_r^t \tag{2.12}$$

$$\therefore\ A_r = A_{r-1} - p_r\, u_r^t - u_r\, p_r^t + u_r\, (u_r^t\, p_r)\, u_r^t/2K_r^2 \tag{2.13}$$

again defining,

$$q_r = p_r - \tfrac{1}{2}\, u_r\ (u_r^t\, p_r/2K_r^2) \tag{2.14}$$

then,

$$A_r = A_{r-1} - q_r\, u_r^t - (q_r\, u_r^t)^t \tag{2.15}$$

The matrix of interest is $(Q_r\, B_{r-1}\, Q_r)$, Equ. (2.10), which is of order n-r, since the elements in the first (r-1) rows and columns are the same as the corresponding elements of A_{r-1}. Therefore, the transformation matrix P_r is designed as to eliminate a_{ri} (i = r+2, ...,n) and to convert $a_{r,r+1}$ to $\pm S_r$, where S_r is defined before. Hence the vector p_r has its first (r-1) elements equal to zero, Equ. (2.12), and the rth element is not needed. Therefore there are $(n-r)^2$ multiplications required for the evaluation of p_r. Similarly, it can be proved that there are (n-r) multiplications involved in the computation of q_r. Finally the total number of multiplications involved in the computations of the elements of A_r is $[2(n-r)^2 + (n-r)]$, but since the matrix is symmetric, the number of multiplications concerning the upper triangle only is $(n-r)^2 + (n-r)/2$, the second term may be neglected compared to the first, $\therefore$ No. of multiplication for one transformation $\simeq$ $(n-r)^2$. Hence, the total number of multiplications to reduce the matrix to a tri-diagonal one is essentially $\frac{2}{3}\, n^3$ which is half of that of Givens' method.

2.3 Computing Eigen-Values and Eigen-Vectors

2.3.1 Eigen-Values

Once the matrix has been reduced to the tri-diagonal form, there are so many techniques for getting the eigen-values, for example (Bisecting method, Sturm sequence or Gerschgorin discs).

An effective method for computing eigen-values that are most suitable for parallel computers can be summarized as follows. The largest and smallest eigen-values can be obtained by the power method. Dividing the interval between $\lambda_{min} = \lambda_1$ and $\lambda_{max} = \lambda_n$ into (n-1) divisions and get λ_j (j=1,2,....,n). For each λ_j, the determinate of the tri-diagonal matrix can be obtained, using the method of principal minors

$$
\begin{aligned}
f_o(\lambda_j) &= 1 \\
f_1(\lambda_j) &= (a_{11} - \lambda_j) \\
f_1(\lambda_j) &= (a_{kk} - \lambda_j)\, f_{k-1}(\lambda_j) - a^2_{k-1,k}\, f_{k-2}(\lambda_j)
\end{aligned}
\qquad (2.16)
$$

supposing that the value of each determinate corresponding to λ_j is R_j (j=1,2,....,n) a plot can be made between λ_j and R_j, Figure a.

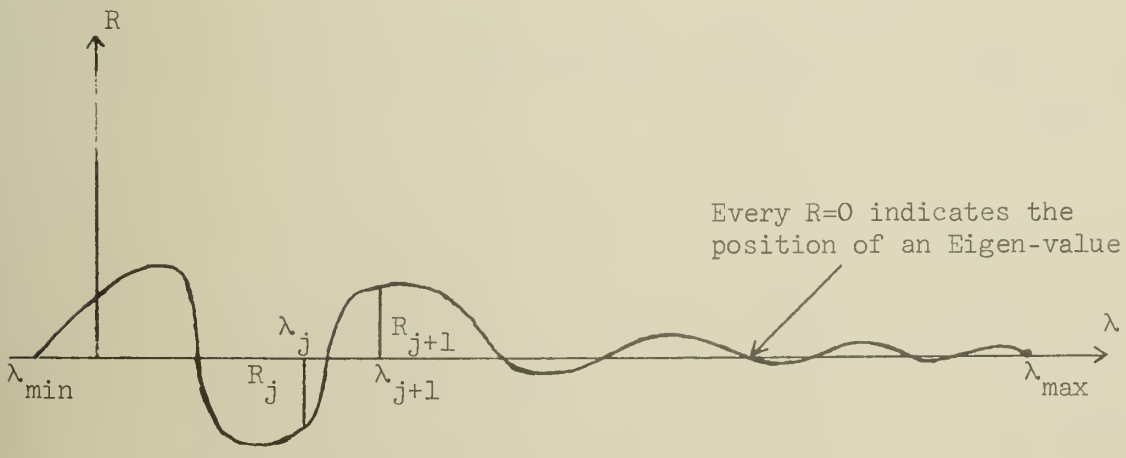


Figure (a)

Each zero crossing indicates a correct eigen-value, supposing that λ_{j+1} gave use to a positive residual R_{j+1} while λ_j gave a negative R_j, then, again the interval between λ_j and λ_{j+1} can be divided to get the exact eigen-value with enough accuracy.

2.3.2 Eigen-Vectors

If $\mathbf{x}$ is the eigen-vector of the tridiagonal matrix (A_{n-2}) corresponding to an eigen-value λ, then the corresponding eigen-vector, ν, of the original matrix A_o is given by,

$$\nu = P_1 P_2 \cdots P_{n-2} x$$

To avoid storing the matrices P_i, the vector ν is calculated from the relations,

$$\begin{aligned} & P_{n-2} x = (x)_{n-2} \\ & P_{n-3}(x)_{n-2} = (x)_{n-3} \\ & \text{- - - - - - - - - -} \\ & \nu = P_1(x)_2 = (x)_1 \end{aligned} \qquad (2.17)$$

So, it remains to find the eigen-vectors of the tri-diagonal matrix. Assume that the tri-diagonal matrix (A_{n-2}) can be expressed in the form,

$$A_{n-2} = \left[\begin{array}{c|c|c|c|c|c|c|c} C_1 & b_1 & 0 & 0 & 0 & 0 & 0 & 0 \\ b_1 & C_2 & \cdot & 0 & 0 & 0 & 0 & 0 \\ 0 & \cdot & \cdot & \cdot & 0 & 0 & 0 & 0 \\ 0 & 0 & \cdot & C_i & b_i & 0 & 0 & 0 \\ 0 & 0 & 0 & b_i & C_{i+1} & b_{i+1} & 0 & 0 \\ 0 & 0 & 0 & 0 & b_{i+1} & \cdot & \cdot & 0 \\ 0 & 0 & 0 & 0 & 0 & \cdot & \cdot & \cdot \\ 0 & 0 & 0 & 0 & 0 & 0 & \cdot & \cdot \end{array}\right]$$

for an eigen-value λ and an eigen-vector x, where $x^t = (x_1, x_2, \ldots, x_n)$, of (A_{n-2}) the following relation can be obtained,

$$(C_1 - \lambda)\, x_1 + b_1\, x_2 = 0$$

$$b_{i-1}\, x_{i-1} + (C_i - \lambda)\, x_i + b_i\, x_{i+1} = 0 \quad (i=2,3,\ldots,n-1)$$

$$b_{n-1}\, x_{n-1} + (C_n - \lambda)\, x_n = 0 \qquad (2.18)$$

If $x_1 = 0$, the rest of the components of the vector x are zero, so x_1 will be assumed as 1. From the equations (2.18) and (2.16) an expression for x_r $(r=2,\ldots,n)$ may be obtained in form,

$$x_r = (-1)^{r-1}\, f_{r-1}(\lambda)/(b_1\, b_2 \cdots b_r) \quad (r=2,\ldots,n) \qquad (2.19)$$

Similarly all the eigen-vectors of the original matrix A_o may be obtained, each corresponding to an eigen-value.

A flow chart and a higher level language program for this algorithm to find all eigen-values are in sections 2.4.

2.4 High Level Language Program

```
#BEGIN
      #LABEL    START, LOOP1, LOOPE, LP, LLP, L1, LL1, L2, SEUP, REPE,
                FINAL, FNAL, EEND;
      #ARRAY    #REAL  #FLOAT  #SHORT  #STRAIGHT
                A[64,64], MQ[64,64], TMQ[64,64], FNT[64,64], FCN[64,64],
                UT[64,1], U[1,64], P[1,64], Q[1,64], EVALUE[64],
                FNTO[64], FNT1[64], FTO[64], FT1[64], NEVAL[64];
      #REAL     #FLOAT  #SHORT
                S2, S, T, TEM, GRAG, INTV, STRT, FINI, RANG, STP,
                SSTP, INIA, FINB;
      #REAL     #FIXED
                J, CNT, NCNT, EV, TK, JK, NUM, NUM1, H, L, K, I, N, M;
      #SET      HH, LL, KK, II, JJ, IJ, JI, MM, NN;
START:  #READ   A;
                J ← 2;
LOOP1:          HH :: [J,J+1, ...., 64];
                LL :: [1, 2, ...., J-1];
                KK :: [J+1, J+2, ...., 64];
                S2 ← #FOR (H) #SIM (HH)#DO #SUM(A[J-1,H]↑2);
                S ← SQRT (S2);
                T ← S2 - A[J-1,J] x S;
                #FOR (L) #SIM (LL)#DO  UT[L,1] ← 0.;
                UT[J,1] ← A[J-1,J] - S;
                #FOR (K)#SIM (KK)#DO UT[K,1] ← A[J-1,K];
                U ← TRANSPOSE[UT];
                P ← A x U;
                P ← P/T;
                TEM ← UT x P;
                TEM ← TEM/T;
                Q ← .5 x TEM x U;
                Q ← P-Q;
```

```
                MQ ← Q x UT;
                MQT ← TRANSPOSE[MQ];
                A ← A - MQ - MQT;
                J ← J+1;
        #IF     J ≤ 63 #THEN  #GOTO  LOOP1;
                NCNT ← 0;
                EV ← 1;
                GRAG ← EMAX[A] - EMIN[A]; (SEE APPENDIX A)
                INTV ← GRAG/8. ;
                STRT ← EMAX[A];
                II :: [1,2,....,64];
                JJ :: [3,4,....,64];
LOOPE:          FINI ← STRT - INTV;
                RANG ← STRT - FINI;
                STP ← RANG/64.;
        #FOR    (I) #SIM (II) #DO
                #BEGIN   EVALUE[I] ← FUNI + STR x (I-1);
                     FNTO[I] ← 1.;
                     FNT1[I] ← A[1,1] - EVALUE[I];
                #END;
        #FOR    (K) #SEQ (JJ) #DO
        #FOR    (I) #SIM (II)#DO
                #BEGIN
                     FNT[K,I] ← (A[K,K]-EVALUE[I]) x FNT1[I] -
                     A[K-1,K]↑2-FNTO[I];
                     FNO[I] ← FNT1[I];
                     FNT1[I] ← FNT[K,I];
                #END
                NCNT ← NCNT + 1;
                TK ← 1;
                CNT ← 0;
```

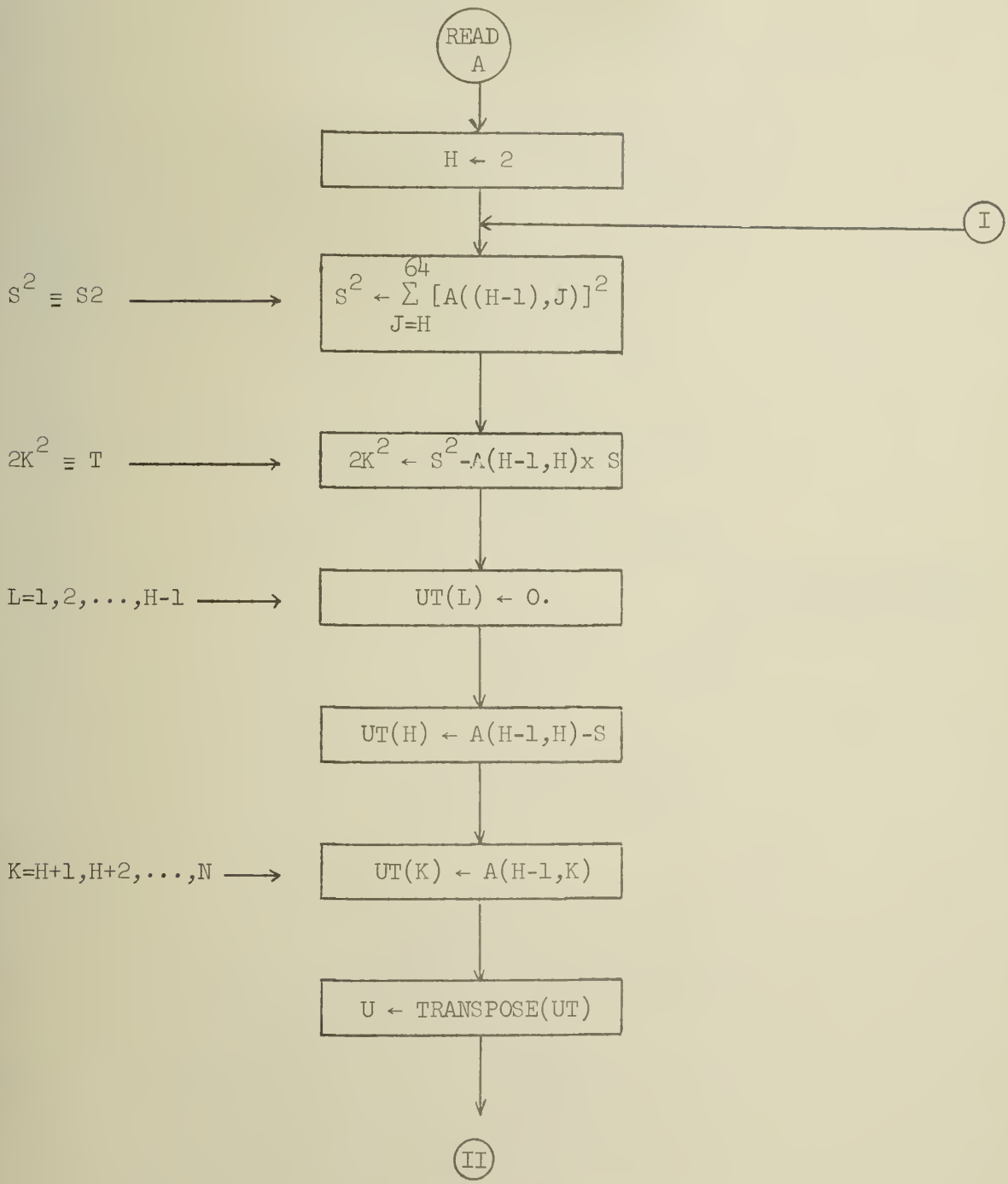
```
LP:     #IF     TK ≥ 64 #THEN  #GOTO  FINAL  #ELSE
        #IF     FNT[64,TK] < 0. #THEN  #GOTO  L1  #ELSE
        #IF     FNT[64,TK] > 0.  #THEN  #GOTO  L2  #ELSE  #DO
                #BEGIN
                     #PRINT  EVALUE [TK];
                     EV ← EV + 1;
                     #IF EV ≥ 63  #THEN  #GOTO  EEND  #ELSE  #DO
                          #BEGIN
                               TK ← TK + 1;
                               #GOTO LP;
                          #END;
                #END;
L1:     #IF     TK ≥ 64  #THEN  #GOTO  FINAL  #ELSE
        #IF     FNT[64,TK+1] < 0. #THEN  #DO
                     #BEGIN
                          TK ← TK+1; #GOTO L1;
                     #END  #ELSE
        #IF     FNT[64,TK+1] > 0.  #THEN  #DO
                     #BEGIN
                          TK ← TK+1; #GOTO SEUP;
                     #END  #ELSE  #DO
                #BEGIN
                     #PRINT EVALUE [TK+1];
                     EV ← EV + 1;
                     #IF EV ≥ 64  #THEN  #GOTO  EEND  #ELSE  #DO
                          #BEGIN
                               TK ← TK+1; #GOTO L1;
                          #END;
                #END;
L2:     #IF     TK ≥ 64  #THEN  #GOTO  FINAL  #ELSE
        #IF     FNT[64,TK+1] < 0.  #THEN  #DO
                     #BEGIN
                          TK ← TK+1; #GOTO  SEUP;
                     #END  #ELSE
```

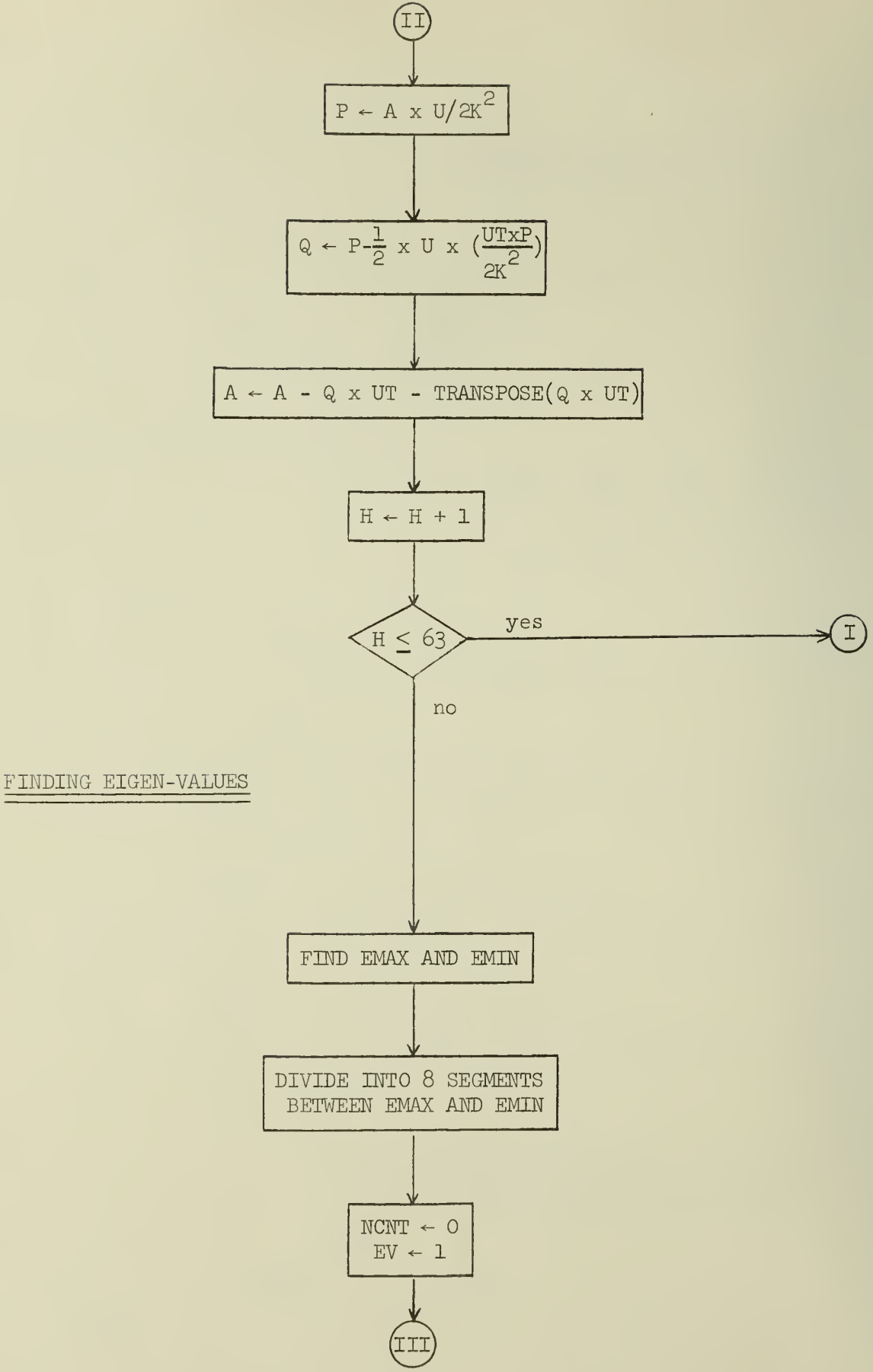
```
          #IF       FNT[64,TK+1] > 0.  #THEN  #DO
                         #BEGIN
                              TK ← TK+1; #GOTO L2;
                         #END  #ELSE  #DO
                         #BEGIN
                              #PRINT EVALUE [TK+1];
                              EV ← EV + 1;
                              #IF EV ≥ 64  #THEN  #GOTO  EEND  #ELSE
                                   #BEGIN
                                        TK ← TK+1; #GOTO  L2;
                                   #END;
                         #END;
SEUP                     NUM ← CNT x 16 + 1;
                         NUM1 ← NUM + 15;
                              NN :: [1,2,....,16];
                              MM :: [NUM, NUM+1,....,NUM1];
                              INIA ← EVALUE [TK-1];
                              FINB ← EVALUE [TK];
                              SSTP ← (FINB - INIA)/16. ;
                         #FOR (N) #SIM (NN) #DO
                         #FOR (M) #SIM (MM) #DO
                              #BEGIN
                                   NEVAL [M] ← INIA + SSTP x (N-1);
                                   FTO[M] ← 1. ;
                                   FT1[M] ← A[1,1] - NEVAL[M];
                              #END;
                         CNT ← CNT + 1;
          #IF            CNT < 4  #THEN  #GOTO  LP  #ELSE
REPE                     TEM ← CNT x 16
                         IJ :: [1,2,....,TEM];
                         JI :: [3,4,....,TEM];
                         #FOR (K) #SEQ (JI) #DO
                         #FOR (I) #SEQ (IJ) #DO
                              #BEGIN
                                   FCN[K,I]← (A[K,K]-NEVAL[I]) x FT1[I] -
                                   A[K-1,K]↑2 x FTO[I];
```

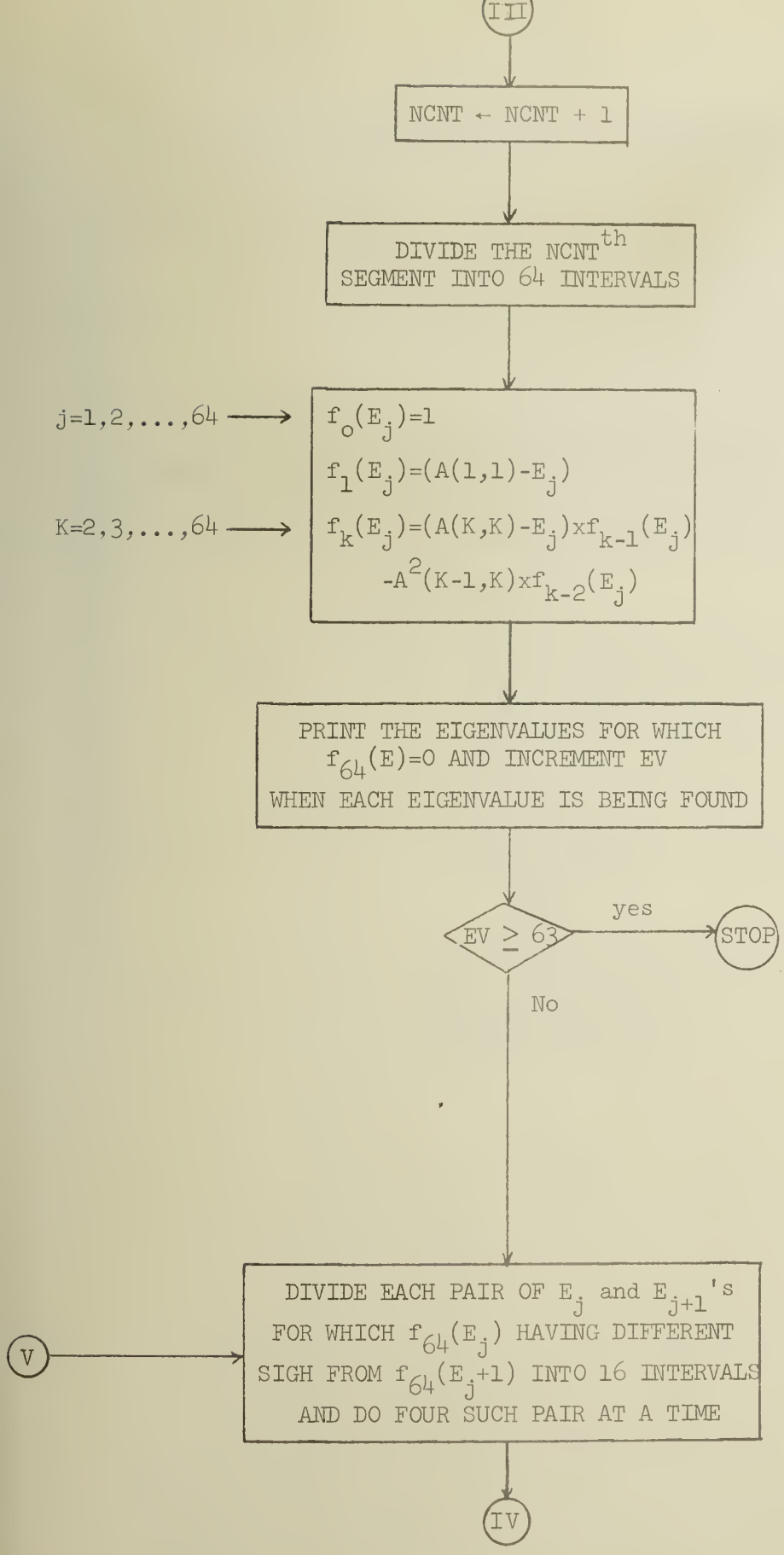
```
                              FTO[I] ← FT1[I];
                              FT1[I] ← FCN[K,I];
                         #END;

                    JK ← 1

LLP :    #IF        JK ≥ 64  #THEN  #GOTO  FINAL  #ELSE

         #IF        FCN[64,JK] = 0.  #THEN  #DO
                         #BEGIN
                              #PRINT NEVAL [JK];
                              EV ← EV + 1;
                              #IF EV ≥ 64  #THEN  #GOTO  EEND  #ELSE  #DO
                                   #BEGIN
                                        JK ← JK + 1; #GOTO LL1;
                                   #END;
                         #END  #ELSE  #DO
                         #BEGIN
                              JK ← JK + 1; #GOTO  LLP;
                         #END;

LL1:     #IF        FCN[64,JK] = 0  #THEN  #GOTO  LL1 #ELSE  #DO
                    #BEGIN
                         JK ← JK+1; #GOTO LLP;
                    #END;

FINAL:   #IF        CNT = 0  #THEN  #GOTO  FNAL  #ELSE  #GOTO  REPE;

FNAL:    #IF        NCNT < 8  #THEN  #DO
                    #BEGIN
                         STRT ← FINI; #GOTO  LOOPE;
                    #END  #ELSE  #DO
                         #BEGIN
                              #PRINT "SOME ROOTS ARE EITHER TOO CLOSE
                              TOGETHER OR THERE EXIST MULTIPLE ROOTS";
                              #STOP;
                         #END;

EEND:    #END
```

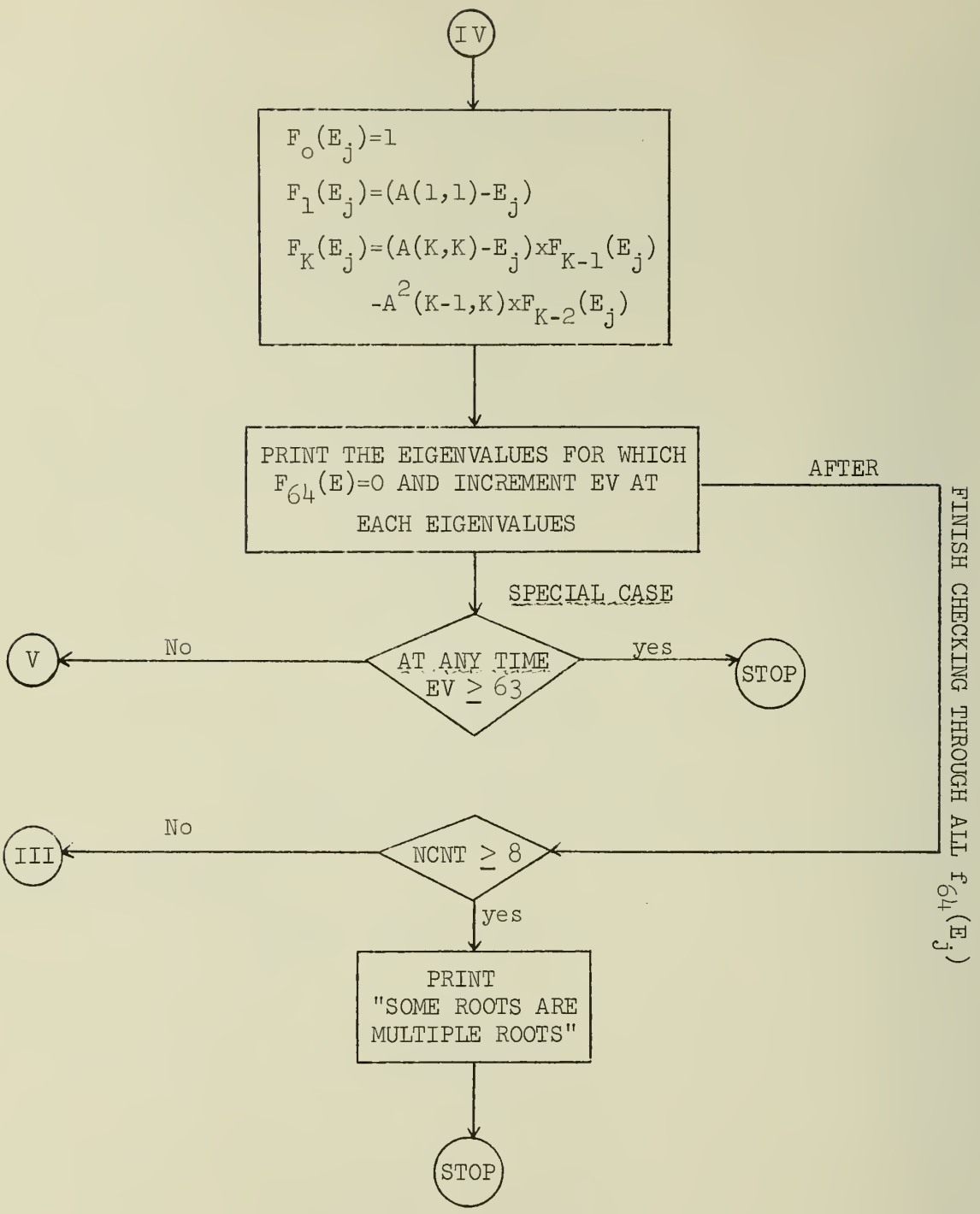
2.5 Flow Chart

HOUSEHOLDER TRIDIAGONALIZATION




II
P ← A x U/2K²
Q ← P-½ x U x (UTxP/2K²)
A ← A - Q x UT - TRANSPOSE(Q x UT)
H ← H + 1
H ≤ 63
yes
I
no
FINDING EIGEN-VALUES
FIND EMAX AND EMIN
DIVIDE INTO 8 SEGMENTS BETWEEN EMAX AND EMIN
NCNT ← 0
EV ← 1
III


III
NCNT ← NCNT + 1
DIVIDE THE NCNT^th SEGMENT INTO 64 INTERVALS
j=1,2,...,64
f_o(E_j)=1
f_1(E_j)=(A(1,1)-E_j)
K=2,3,...,64
f_k(E_j)=(A(K,K)-E_j)xf_k-1(E_j)
-A^2(K-1,K)xf_k-2(E_j)
PRINT THE EIGENVALUES FOR WHICH f_64(E)=0 AND INCREMENT EV WHEN EACH EIGENVALUE IS BEING FOUND
EV ≥ 63
yes
STOP
No
V
DIVIDE EACH PAIR OF E_j and E_j+1's FOR WHICH f_64(E_j) HAVING DIFFERENT SIGH FROM f_64(E_j+1) INTO 16 INTERVALS AND DO FOUR SUCH PAIR AT A TIME
IV


IV
$F_o(E_j)=1$
$F_1(E_j)=(A(1,1)-E_j)$
$F_K(E_j)=(A(K,K)-E_j)xF_{K-1}(E_j)$
$-A^2(K-1,K)xF_{K-2}(E_j)$
PRINT THE EIGENVALUES FOR WHICH $F_{64}(E)=0$ AND INCREMENT EV AT EACH EIGENVALUES
AFTER FINISH CHECKING THROUGH ALL $f_{64}(E_j)$
SPECIAL CASE
AT ANY TIME EV $\geq$ 63
No
yes
V
STOP
NCNT $\geq$ 8
No
III
yes
PRINT "SOME ROOTS ARE MULTIPLE ROOTS"
STOP

3. An Algorithm For Finding Eigen-values and Eigen-vectors For Non-Symmetric Matrices

3.1 Introduction

There are two algorithms for dealing with unsymmetric matrices. The first one is (LR) which was developed by Rutishauser (1958). The LR-algorithm has proved to be a powerful method for finding the eigen-values of symmetric band matrices. However, it proved to be inferior for handling the eigen-value problem of large non-symmetric matrices, the most important difficulties are,

(i) Triangular decomposition may not always be possible.

(ii) The amount of computation required by the method is very great. The LR-algorithm is essentially an iterative procedure, where the method consists of a series of similarity transformations on the original matrix,

$$A_1 = L_1 R_1 \tag{3.1}$$

where,

A_1 is the matrix under consideration (size $\equiv$ n).

L is a lower triangular matrix.

R is an upper triangular matrix

i.e.,

$$L = \begin{bmatrix} \ell_{1,1} & & 0 \\ & \ddots & \\ \ell_{n,1} & & \ell_{n,n} \end{bmatrix} \quad \text{where } \ell_{ii} = 1,\ i = 1,\ n$$

$$R = \begin{bmatrix} r_{11} & \cdots & r_{1n} \\ & \ddots & \\ 0 & & r_{nn} \end{bmatrix}$$

by similarity transformation,

$$A_2 = L_1^{-1} A_1 L_1 \tag{3.2}$$

hence $A_2 = R_1 L_1$

similarly

$$A_{k+1} = (L_k^{-1} L_{k-1}^{-1} \cdots L_1^{-1})(A_1)(L_1 L_2 \cdots L_k)$$

hence

$$(L_1 L_2 \cdots L_k) A_{k+1} = A_1 (L_1 L_2 \cdots L_k) \tag{3.3}$$

let, $T_k = L_1 L_2 \cdots L_k$

$$U_k = R_k R_{k-1} \cdots R_1$$

therefore, $T_k U_k = L_1 L_2 \cdots L_{k-1} (L_k R_k) R_{k-1} \cdots R_2 R_1$

$$= A_1 L_1 L_2 \cdots L_{k-1} R_{k-1} \cdots R_2 R_1$$

and so forth,

finally $$T_k U_k = (A_1)^k \tag{3.4}$$

and it was shown that, if certain conditions are fulfilled then, as $k \to \infty$, then $(A_1)^k$ tends to an upper-triangular matrix, where the diagonal elements are the eigenvalues in order of their modulus, (the first being the largest), i.e.,

$$(A_1)^k = \begin{bmatrix} \lambda_1 & & \times \\ & \lambda_2 & \\ \bigcirc & & \ddots \lambda_n \end{bmatrix} \tag{3.5}$$

The restrictions on the LR-algorithm may be summarized as follows:

1. Eigenvalues must be distinct since convergence depends upon the ratio $(\lambda_{r+1}/\lambda_r)$ and will be very slow if

separation of the eigenvalues is poor.

2. Triangular decomposition of A must exist, (it fails if any of the principal minors of A vanishes).

3. The leading principal minors of the modal matrix, (matrix whose columns are the eigenvectors), and its inverse should not vanish.

4. Even if the triangular decomposition exists, there is a large class of matrices where triangular decomposition is numerically unstable which may lead to gross errors in the values of the computed eigenvalues.

5. The volume of computation is very large, $\frac{2}{3} n^3$ multiplications for each step of iteration, half of them in triangulation and the other half in post-multiplication.

Due to the above mentioned restriction on LR, QR proved to be a much more efficient algorithm, and it will be discussed here in more detail.

A more stable method of triangulation may be achieved by using elementary unitary transformations (QR-algorithm). The QR-algorithm is defined by the relations,

$$A_k = Q_k R_k$$
$$A_{k+1} = Q_k^{-1} A_k Q_k = R_k Q_k \qquad (3.6)$$

where R is an upper-triangular matrix,

Q is a unitary matrix, i.e.,

$Q^t = Q^{-1}$

The unitary-triangular decomposition of any square matrix exists.

3.2 Theoretical Background (QR-Algorithm)

(1) For any matrix A (order n), there exists a unitary matrix Q such that

$$A_k = Q_k R_k$$

where, R_k is an upper triangular matrix which has real positive diagonal elements, and $Q_k = N_1 N_2 \cdots N_n$ (3.7)

where

$$N_i = \begin{bmatrix} I_{i-1} & 0 \\ 0 & M_i \end{bmatrix} \qquad (3.8)$$

M_i is of order [n-(i-1)], the unitary matrix M can operate on any vector b such that,

$$M^t b = ||b|| \, e_1 \qquad (3.9)$$

where

$||b||$ is the Eulerian norm of the vector b, $e_1 = \{1,0,0,\ldots 0\}$

M is an elementary unitary matrix, that is a matrix which differs from an identity matrix at most in one principal 2 x 2 submatrix. Therefore, if the vector b is of order m, $M^t = T_m T_{m-1} \cdots T_2 T_1$ (3.10)

where,

$$T_r = \begin{bmatrix} t_{ll} & 0 \cdots 0 & t_{lr} & \\ 0 & 1 \cdots 0 & 0 & \\ \vdots & \ddots & \vdots & 0 \\ 0 & 0 \cdots 1 & 0 & \\ t_{rl} & 0 \cdots 0 & t_{rr} & \\ & 0 & & \begin{matrix} 1 & \\ & \ddots \\ & & 1 \end{matrix} \end{bmatrix}$$

The elements of T_r can be easily expressed in terms of the components of vector b,

$$(t_{11})^r = (t_{rr})^r = [1 - \frac{|br|^2}{\sum_{p\leq r} |b_p|^2}]^{\frac{1}{2}}$$
$$(t_{1r})^r = -(t_{r1})^r = b_r / \sum_{p\leq r} |b_p|^2 \qquad (3.11)$$

Hence, if we consider that the matrix A_k consists of the column vectors C_j, $j = 1, \ldots, n$, therefore,

$$N_n^t \, N_{n-1}^t \, \ldots \, N_2^t \, N_1^t \, A_k = R_k \qquad (3.12)$$

where N_i is defined by Eq. (3.8), and $r_{ii} = ||b_i||$, where

$$b_i = \{a_{ii}, a_{i+1,i}, \ldots, a_{n,i}\}, \; i = 1, \ldots, n$$

(2) Before starting this step, it is of importance to prove that Q is unique if A is non-singular.

Proof: Suppose that

$$A = Q_1 R_1 = Q_2 R_2$$

therefore, if A is non-singular, R_1 and R_2 are non-singular

$$R_1 R_2^{-1} = Q_1^{-1} Q_2 = Q_1^t Q_2$$

therefore, $(R_1 R_2^{-1})$ is unitary also, hence, $(R_1 R_2^{-1})^{-1} = (R_1 R_2^{-1})^t$ (3.13)
since $R_1 R_2^{-1}$ is an upper-triangular matrix, Eq. (3.13) is satisfied if and only if $R_1 R_2^{-1}$ is a diagonal matrix, therefore, $R_1 R_2^{-1} = I$, i.e. $R_1 = R_2$, hence $Q_1 = Q_2$.

Step (1) is completed by post multiplying (R_k) from Eq. (3.12) by $Q = N_1 N_2 \cdots N_n$ to get $A_{k+1} = R_k Q_k$ and the process is repeated all over again until all the elements a_{ij}, $i > j$ of the matrix A vanish if the eigenvalues of A are distinct. Similar to the LR-algorithm the process may be looked upon as follows,

$$A_k = Q_k R_k$$

$$A_{k+1} = Q_k^t A_k Q_k = R_k Q_k$$

if

$$P_k = Q_1 Q_2 \cdots Q_k$$

$$S_k = R_k R_{k-1} \cdots R_2 R_1$$

therefore,

$$P_k S_k = Q_1 Q_2 \cdots Q_{k-1} (Q_k R_k) R_{k-1} \cdots R_2 R_1$$

$$= A_1 Q_1 Q_2 \cdots Q_{k-2}(Q_{k-1} R_{k-1}) R_{k-2} \cdots R_2 R_1$$

and so forth, finally

$$P_k S_k = (A_1)^k \qquad (3.14)$$

Since P_k is unitary and S_k is an upper-triangular, therefore the unitary-triangular decomposition of a non-singular matrix is unique, according to the lemma proved at the beginning of step (2). Francis (1961) proved the convergence of $P_k S_k = (A)^k$ and showed that, "If any non-singular matrix A has eigenvalues of distinct modulus, then under the QR transformation the elements below the principal diagonal tend to zero, the moduli of those above it tend to fixed values, and the elements on the principal diagonal itself tend to the eigenvalues."

However, for eigenvalues of the equal modulus, it can be shown that the matrix A_k becomes split into independent principal submatrices coupled only in so far as the eigenvectors are concerned. Usually, each of these submatrices is associated with groups of eigenvalues of the same modulus.

(3) The amount of computations that the method involves in one iteration is proportional to n^3. However if the matrix is first reduced to the upper Hessenberg form by any stable unitary transformation technique, i.e., Householder's method, the amount of computations become proportional to n^2. Reducing the matrix to the upper Hessenberg form has some advantages,

(i) If any element a_{ij} of the matrix A is zero, the matrix can be partitioned at this element and the rest of the submatrices can be operated on separately so far as the eigenvalues are concerned.

(ii) The upper-Hessenberg form is preserved under the QR-transformation.

(iii) If an almost triangular matrix A is non-singular and has non-zero elements, then under the QR-transformation the principal diagonal elements of A_k converge to the eigenvalues in order of size, if they are not equal.

(iv) For distinct eigenvalues, the elements $(a_{ij})_k$ $i > j$, below the principal diagonal converge to zero as $(\frac{\lambda_i}{\lambda_j})^k$. This is the same rate of convergence as the LR-transformation which is rather slow. Taking a closer look at the process where, for example, (a_{nn}) approaches the exact value of the eigenvalue λ_n, it is found that

$$\epsilon_{k+1} = \epsilon_k \cdot \frac{\lambda_n}{\lambda_{n-1}}$$

where $\epsilon_k = (a_{nn})_k - \lambda_n$ and $\frac{\lambda_n}{\lambda_{n-1}} < 1$

To speed up convergence, the origin of all eigenvalues is shifted by the value ζ_k before an iteration and then shifted back again after the iteration, therefore

$$\epsilon_{k+1} = \epsilon_k \cdot \frac{\lambda_n - \zeta_k}{\lambda_{n-1} - \zeta_k}$$

which is quite an improved rate of convergence for $\zeta_k \tilde{=} \lambda_n$.

Therefore, the generalized QR-transformation becomes,

$$A_{k+1} = Q_k^* A_k Q_k$$

where $$Q_k^* (A_k - \zeta_k I) = R_k$$

In Section 3 methods of choosing the origin shifts ζ_k are discussed for both distinct and equal modulus eigenvalues especially conjugate complex pairs. Once λ_n is found by sufficient accuracy, the order of the matrix may be reduced by omitting the last row and column.

3.3 Numerical Solution for QR Algorithm

The numerical solution for Q-R algorithm can be divided into two major steps, viz.,

(a) Applying Householder's technique to reduce a matrix to an upper Hessenberg form.

(b) Using unitary transformation with the help of origin-shift to iterate, i.e.,

$$A^{(1)} = A$$

$$A^{(K+1)} = Q^{(K)t} A^{(K)} Q^{(K)}$$

where $Q^{(K)t} A^{(K)}$ produces an upper-diagonal matrix.

Since the Householder's technique has been discussed in detail already, therefore we will only concentrate on part (b).

The very first step in part (b) is to determine the value of the origin shift. To find this value, we first find the roots of the last principal 2 x 2 submatrix and then distinguish the following two cases.

(1) Roots are real: choose one that differs least from the last diagonal element $a_{nn}^{(k)}$ (before kth iteration) and call this root $\lambda^{(k)}$, then compare $\lambda^{(k)}$ with the previous one, viz. $\lambda^{(k-1)}$ (assuming $\lambda^{(0)}=0$) if

$$\left| \frac{\lambda^{(k)} - \lambda^{(k-1)}}{\lambda^{(k)}} \right| < \frac{1}{2}$$

set $E^{(k)} = \lambda^{(k)}$, otherwise set $E^{(k)} = 0$ where $E^{(k)}$ is the value of origin shift for kth iteration.

(2) Roots are complex: we do one of two things,

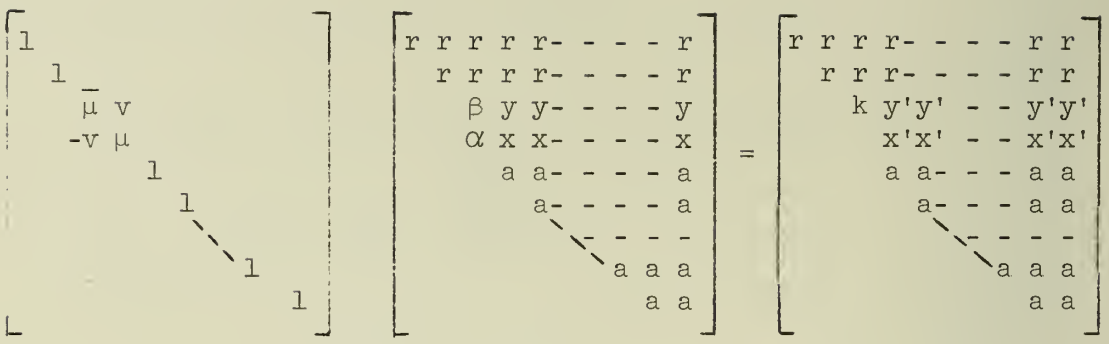
(i) Set $\lambda^{(k)} = |R|$, where $|R|$ is the modulus of the complex roots and then compare with $\lambda^{(k-1)}$.

(ii) Do double origin-shift by setting $\lambda_1^{(k)} = R$ and $\lambda_2^{(k)} = \bar{R}$, compare with the previous $\lambda_1^{(k-1)}$ and $\lambda_2^{(k-1)}$, (assuming $\lambda_1^{(0)} = 0$, $\lambda_2^{(0)} = 0$).

After the value of origin shift has been found, we subtract all diagonal elements from this value and then start iteration; add the value back after each iteration, until the last subdiagonal element $\to 0$, then deflate the matrix by taking out the last row and column, the last diagonal element is one of the eigen-values.

In doing QR iteration, we simply produce a series of 2 x 2 submatrices to operate on two rows at a time on matrix A and eliminate one subdiagonal element at each time during the pre-multiplication. operate on two columns at a time during the post-multiplication according to the following algorithm:

(i) PRE-multiply:



where the elements r, r' and k are those of the final triangular matrix and the elements a, x, and α are in their original state. The elements in the two rows that are changed are given by:

$$k = (|\beta|^2 + |\alpha|^2)^{\frac{1}{2}}$$

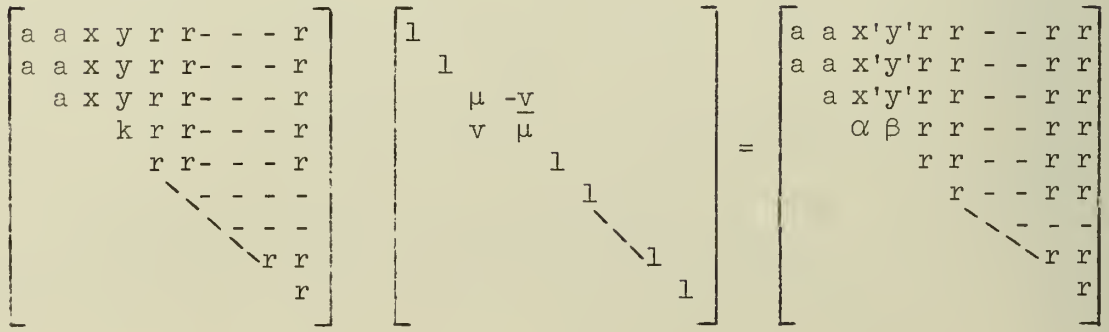
$$x' = \mu x - \nu y$$

$$y' = \bar{\mu} y - \nu x$$

$$\mu = \beta / k$$

$$\nu = \alpha / k$$

(ii) POST-multiply:



Where the elements r, y and k are so far unaltered from the triangular state, and the elements a, x' and α are in their final state. The two columns that are changed are given by:

$$\alpha = vk$$

$$\beta = \bar{\mu}k$$

$$x' = \mu x + vy$$

$$y' = \bar{\mu}y - vx$$

The higher level language program with a detailed flow chart for single origin-shift is in section 3.4 and 3.5.

3.4 High-Level Language Program

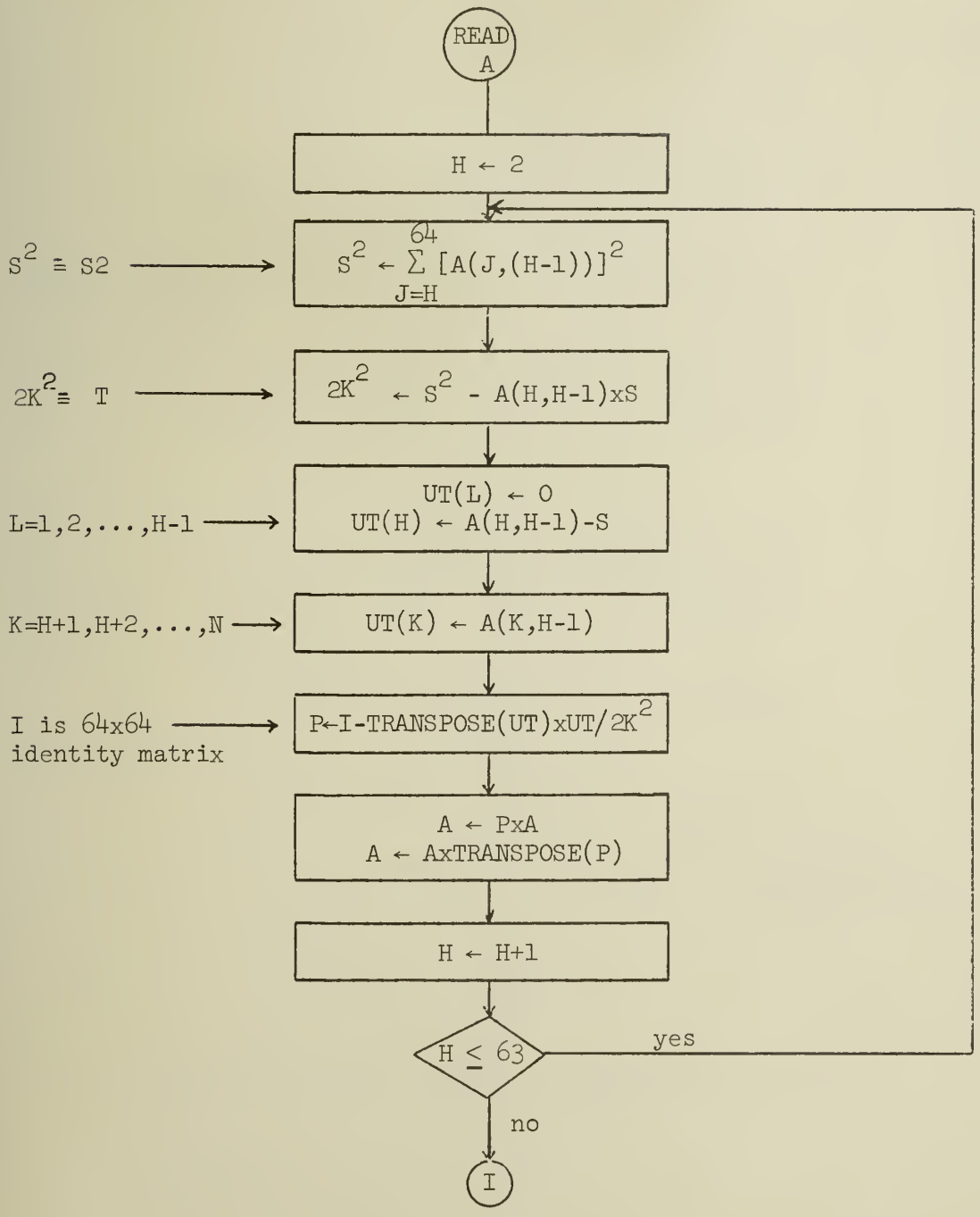
```
#BEGIN
      #LABEL     START, LOOP1, LOOP2, LOOP3, L1P, L2P, L3P, L4P,
                    LOPM, LOPN;
      #ARRAY     #REAL  #FLOAT  #SHORT  #SKWD
                 A[64,64], P[64,64], IM[64,64], UT[64,1], TEM[64,1],
                    V[64], U[64];
      #REAL      #FLOAT  #SHORT
                 S2, S, T, E1, TEMP1, X1, X2, L1, L2, E2, E, K;
      #REAL      #FIXED
                 J, H, L, K1, N, M, I;
      #SET       HH, LL, KK, II, IJ, JI;
START:  #READ    A;
                 J ← 2;
LOOP1:           HH:: [J, J+1, ...., 64];
                 LL:: [1, 2, ...., J-1];
                 KK:: [J+1, J+2, ...., 64];
                 S2 ← #FOR (H) #SIM (HH) #DO  #SUM(A[H,J-1]↑2);
                 S ← SQRT(S2);
                 T ← S2 - A[J,J-1] x S;
                 #FOR (L)  #SIM (LL)  #DO  UT[L,1]←0.;
                 UT[J,1] ← A[J,J-1] - S;
                 #FOR (K1) #SIM (KK)  #DO  UT[K,1] ← A[K,J-1];
                 P ← IM - TRANSPOSE(UT) x UT/T;
                 A ← P x A;
                 A ← A x TRANSPOSE(P);
                 J ← J+1;
        #IF      J ≤ 63  #THEN  #GOTO  LOOP1;
                 N ← 64;
```

```
LOOP2:          II:: [1, 2, ...., N]
                E1 ← 0.;
LOOP3:          TEMP1 ← SQRT((A[N,N] + A[N-1,N-1])↑2-4 x(A[N-1,N-1] x
                   A[N,N] - A[N,N-1] x A[N-1,N]));
                X1 ← .5 x (A[N,N] + A[N-1,N-1] + TEMP1);
                X2 ← .5 x (A[N,N] + A[N-1,N-1]-TEMP1);
                L1 ← ABS(A[N,N] - X1);
                L2 ← ABS(A[N,N] - X2);
        #IF     L1 ≤ L2 #THEN #DO #BEGIN
                E2 ← L1;
                #GOTO L1P;
                #END;
                E2 ← L2;
L1P:    #IF     ABS((E2-E1)/E2) < .5 #THEN #DO #BEGIN
                E ← E2;
                E1 ← E;
                #GOTO L2P;
                #END;
                E ← 0.;
L2P:            #FOR (I) #SIM (II) #DO
                A[I,I] ← A[I,I] - E;
                M ← 1;
LOPM:           JI:: [M+1,M+2,....,M];
                K ← SQRT(A[M,M] ↑2 + A[M+1,M] ↑2);
                U[M] ← A[M,M]/K;
                V[M] ← A[M+1,M]/K;
                #FOR (I) #SIM (JI) #DO #BEGIN
                   TEM[M,I] ← U[M] x A[M,I] - V[M] x A[M+1,I];
                   A[M+1,I] ← U[M] x A[M+1,I] + V[M] x A[M,I];
                   A[M,I] ← TEM[M,I];
                   #END;
                   A[M,M] ← K;
                   A[M+1,M] ← 0.;
                #IF M ≥ N-1 #THEN #GOTO L3P;
```

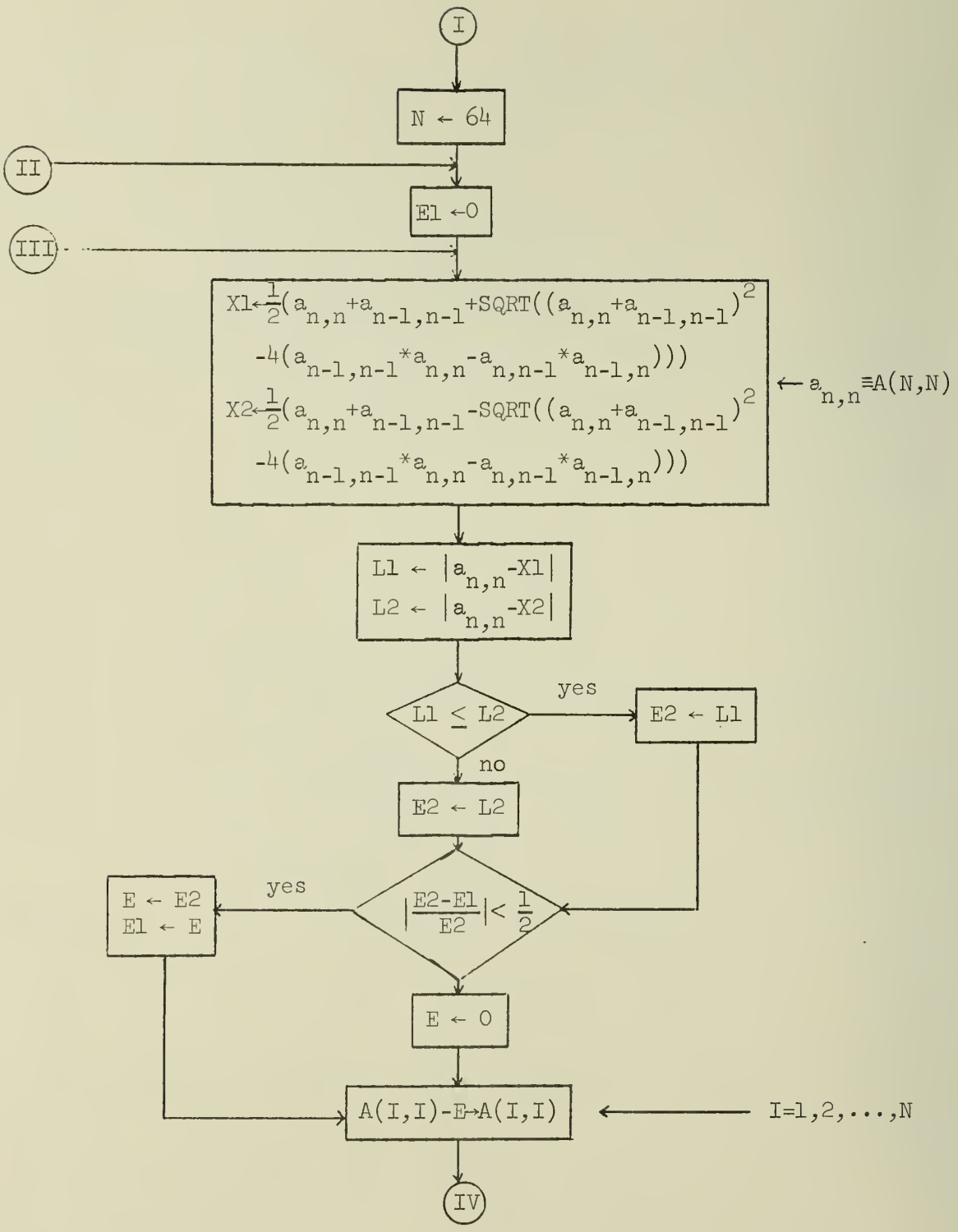
```
                    M ← M+1;
                    #GOTO  LOPM;
L3P:                M ← 1;
LOPN:               IJ:: [1,2,....,M];
                    A[M+1,M] ← V[M] x A[M+1,M+1];
                    A[M+1,M+1] ← U[M] x A[M+1,M+1];
                    #FOR (I) #SIM (IJ) #DO #BEGIN
                       TEM[I,M] ← U[M] x A[I,M] + V[M] x A[I,M+1];
                       A[I,M+1] ← U[M] x A[I,M+1] + V[M] x A[I,M];
                       A[I,M] ← TEM[I,M];
                       #END;
                       M ← M+1;
                    #IF  M ≥ N  #THEN  #GOTO  L4P;
                       #GOTO  LOPN;
L4P:                #FOR (I) #SIM (II)  #DO
                       A[I,I] ← A[I,I] + E;
                    #IF ABS(A[N,N-1]) > .OOOOO1  #THEN  #GOTO LOOP3;
                       #PRINT A[N,N];
                       N ← N-1;
                    #IF N ≥ 1  #THEN  #GOTO  LOOP2;
                       #PRINT A[1,1];
     #END
```

3.5 Flow Chart

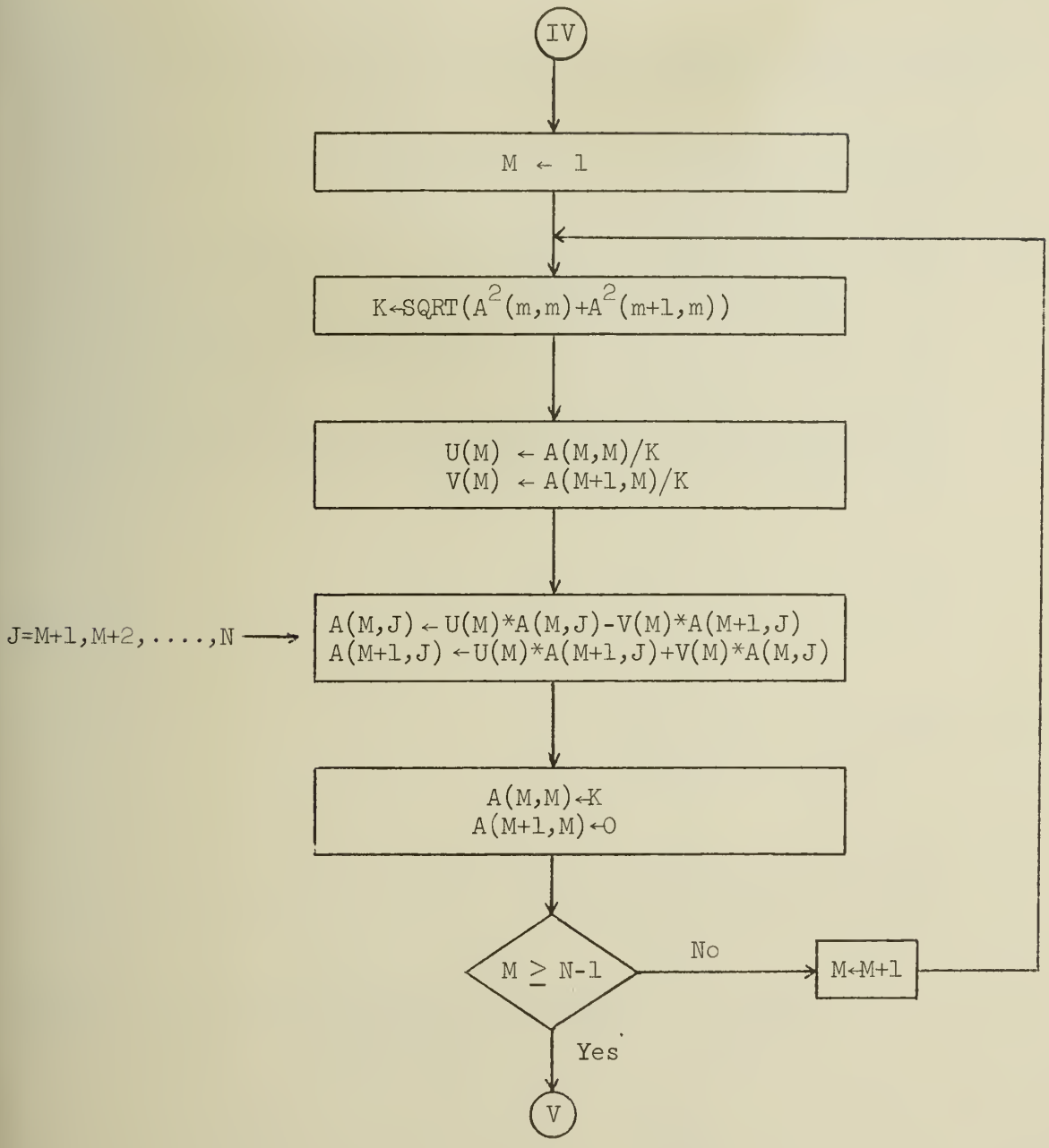
TO GET UPPER HESSENBERG FORM



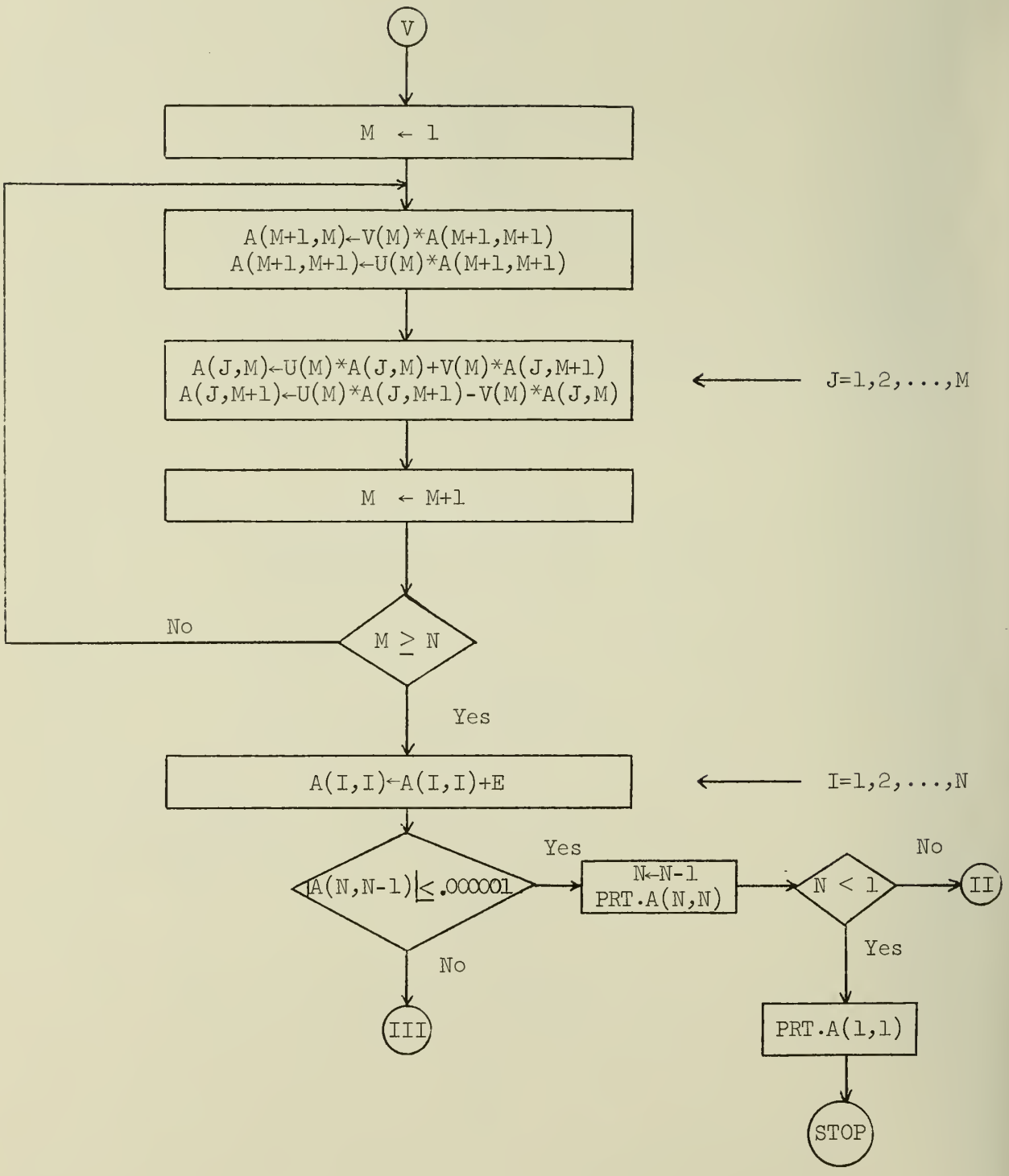
FIND ORIGIN-SHIFT VALUE



PREMULTIPLY: GET UPPER DIAGONAL MATRIX



POST-MULTIPLY AND TESTING



4. References

Bodewig, E., 1959, "Matrix Calculus". Interscience Publishers, New York; North-Holland Publishing Company, Amsterdam.

Francis, J. G. F., 1961, 1962, "The QR Transformation, Part I and II." Computer J. 4, 265-271, 332-345.

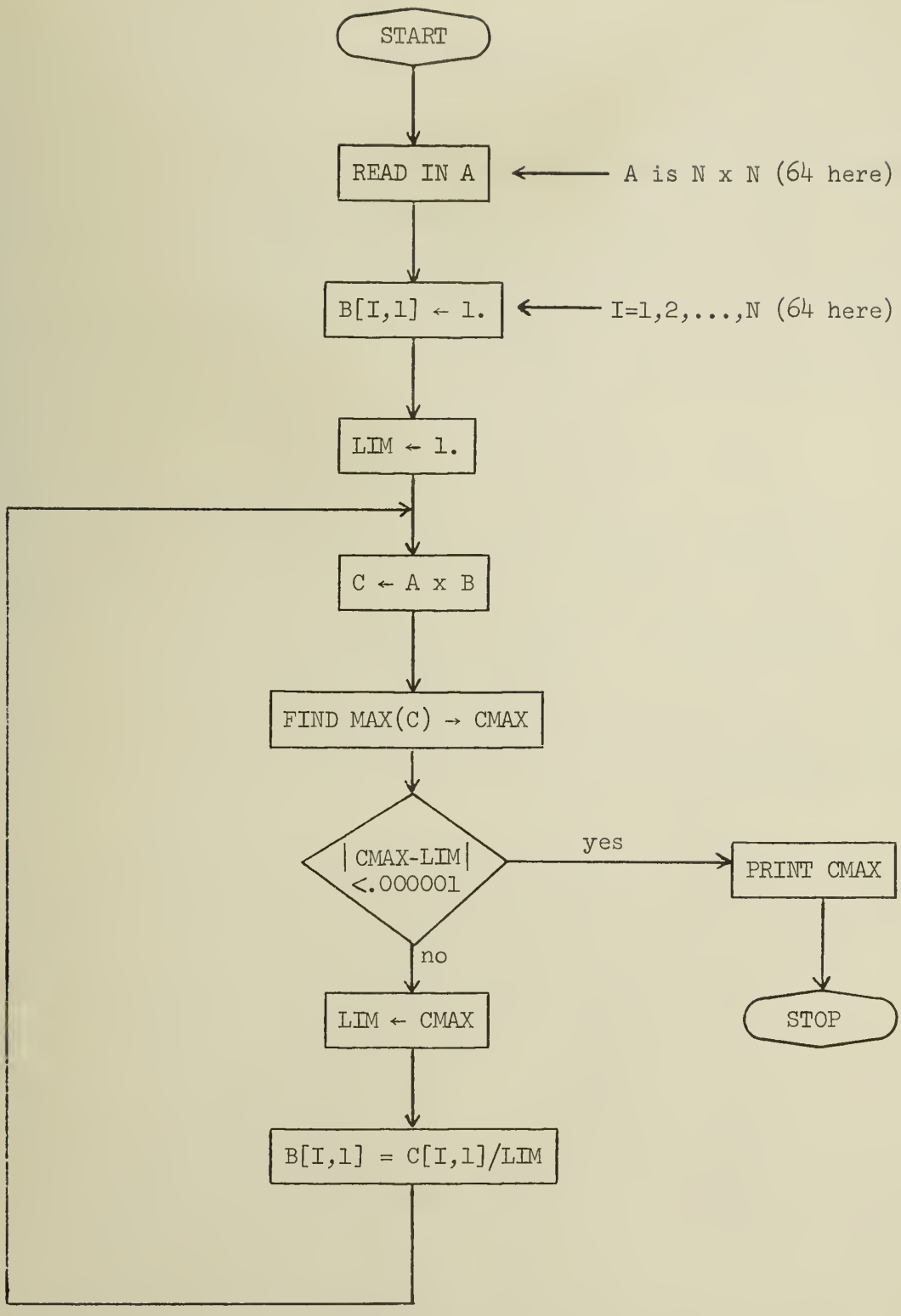
Wilkinson, J. H., 1960a, "Householder's Method for the Solution of the Algebraic Eigenproblem." Computer J. 3, 23-27.

Wilkinson, J. H., 1965, "The Algebraic Eigenvalue Problem." Clarendon Press, Oxford.

Young, David M. Jr., 1960, "Numerical Methods for Solving Problems in Linear Algebra." The University of Texas Computer Center, Austin.

High-Level Language Program For Finding EMAX

```
      #BEGIN
            #LABEL   START, LOOP, LEND;
            #ARRAY   #REAL  #FLOAT  #SHORT  #SKWD
                  A[64,64], B[64, 1], C[64, 1];
            #REAL  #FLOAT  #SHORT
                  LIM, CMAX;
            #REAL  #FIXED
                  I;
            #SET  II, JJ;
START:         #READ A
               II:: [1,2,....,64];
               #FOR (I)  #SIM (II)  #DO
                  B[I,1] ← 1. ;
                  LIM ← 1. ;
LOOP:             C ← A x B;
                  CMAX ← C[1,1];
                  JJ:: [1,2,....,63];
                  #FOR (I)  #SEQ (JJ)  #DO  #BEGIN
                  #IF  ABS(CMAX) - ABS(C[I+1,1]) ≥ 0  #THEN  #GOTO  LEND
                  #ELSE  CMAX ← C[I+1,1];
LEND:             #END;
            #IF  ABS(ABS(CMAX) - ABS(LIM)) ≥ .00001  #THEN  #DO  #BEGIN
                  LIM ← CMAX;
                  #FOR (I)  #SIM (II)  #DO
                  B[I,1] ← C[I,1]/LIM;
                  #GOTO  LOOP;
                  #END  #ELSE
                  #PRINT  CMAX;
      #END
```


START
READ IN A
A is N x N (64 here)
B[I,1] ← 1.
I=1,2,...,N (64 here)
LIM ← 1.
C ← A x B
FIND MAX(C) → CMAX
|CMAX-LIM| <.000001
yes
PRINT CMAX
STOP
no
LIM ← CMAX
B[I,1] = C[I,1]/LIM

APPENDIX B

QR-Algorithm With Double Origin Shift

When a matrix has a complex-pairs of eigen-values, the double origin shift is needed. In this scheme, we combine two iterations into one with a pair of origin shifts since single iteration may not be sufficient enough due to the existence of a complex-pair eigen-values. So at k^{th} stage, we will have

$$A^{(k+2)} = Q^{(k+1)*}\ Q^{(k)*}\ A^{(k)}\ Q^{(k)}\ Q^{(k+1)}$$

Let $W = Q^{(k)}\ Q^{(k+1)}$ and $\Gamma = (A^{(k)} - E^{(k)}I)\ (A^{(k)} - E^{(k+1)}I)$ where $E^{(k)}$ and $E^{(k+1)}$ are the pair of origin shifts, such that $W^*\Gamma$ gives an upper triangular matrix. W^* here is unitary and $W = N_1\ N_2 \ldots N_n$

$N_i = \begin{bmatrix} I_{i-1} & 0 \\ 0 & M_i \end{bmatrix}$ in the previous discussion. Γ has the following form:

$$\begin{bmatrix} x & x & x & x & x & - & - & x \\ x & x & x & x & x & - & - & x \\ x & x & x & x & x & - & - & x \\ & x & x & x & x & - & - & x \\ & & x & x & x & - & - & x \\ & & & x & x & - & - & x \\ & & & & \diagdown & & & \\ & & & & & x & x & x \end{bmatrix}$$

Let the first column be r_1 then N_1 should be defined so that

$$N_1^*\ r_1 = ||r_1|| \cdot e_1$$

Hence, $N_1^* A^{(k)} N_1$ takes the form:

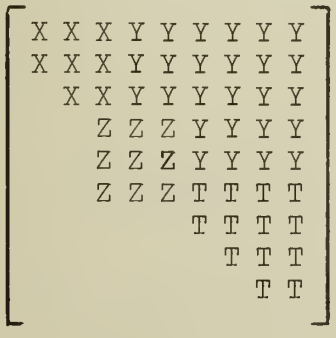
Where the new matrix has few more elements then before, and this is not what we want to do, therefore the general technique will be as follows:

(1) Perform an initial transformation ($N_1^* A^{(k)} N_1$) on $A^{(k)}$ with double origin shifts.

(2) Reduce the resulting matrix to almost triangular form by a method, such as Householder's to obtain $A^{(k+2)}$.

A typical stage in the iteration will take the forms:

Before Householder's transformation:



(ii) After Householder's transformation:

where 3 rows and 3 columns being affected. X and T are those of the initial and final matrices $A^{(k)}$ and $A^{(k+2)}$ respectively, some of the Y's and Z's are changed by the transformation.

Since the first transformation of $A^{(k)}$ by N_1 is similar to the subsequent transformations, we can set up the initial condition for the iteration by finding r_{11}, r_{21}, and r_{31} given by

$$r_{11} = a_{11}(a_{11} - \sigma) + a_{12} \cdot a_{21} + \rho$$

$$r_{21} = a_{21}(a_{11} + a_{22} - \sigma)$$

$$r_{31} = a_{21} \cdot a_{32}$$

where $\sigma = E_1 + E_2$

$\rho = E_1 \cdot E_2$

the transformation matrices of Householder's method are of the form:

$$N_i = N_i^* = I - 2t_i t_i^*$$

t_i is such that $||ti|| = 1$

$t_{ij} = 0$ for $j < i$ and $j > i+2$

$$t_{ii} = \{\frac{1}{2}(1 + \frac{Z_{10}}{k})\}^{\frac{1}{2}}$$

$$t_{i,i+1} = \frac{Z_{20}}{2kt_{ii}}$$

$$t_{i,i+2} = \frac{Z_{30}}{2kt_{ii}}$$

where $k = (\text{sign } Z_{10}) \cdot (Z_{10}^2 + Z_{20}^2 + Z_{30}^2)^{\frac{1}{2}}$

where $Z_{rs} = \begin{bmatrix} Z_{10} & Z_{11} & Z_{12} \\ Z_{20} & Z_{21} & Z_{22} \\ Z_{30} & Z_{31} & Z_{32} \end{bmatrix}$ corresponds to the Z's of the figure above.

In order to reduce the number of multiplication in the Householder's transformation let $H_i^* = [0,\ldots,0,1,X_1,X_2,0,\ldots,0] = \frac{1}{t_{ii}} \cdot t_i^*$

obtaining $N_i = I - (1 + \varphi)\, H_i\, H_i^*$

$$\varphi = \frac{Z_{10}}{k},\quad X_1 = \frac{Z_{20}}{k+Z_{10}},\quad X_2 = \frac{Z_{30}}{k+Z_{10}} \text{ and all} < 1$$

Hence at i^{th} stage, the operation on columns a_i, a_{i+1} and a_{i+2} is $[\underline{a_i}, \underline{a_{i+1}}, \underline{a_{i+2}}]$

$[a_i, a_{i+1}, a_{i+2}] \cdot [I - (1 + \varphi) \begin{bmatrix} 1 \\ X_1 \\ X_2 \end{bmatrix} [1, X_1, X_2]]$ Let $\eta = (1 + \varphi) \cdot$

$\{a_i + X_1\, a_{i+1} + X_2\, a_{i+2}\}$

therefore, $\underline{a_i} = a_i - \eta$

$\underline{a_{i+1}} = a_{i+1} - X_1 \cdot \eta$

$\underline{a_{i+2}} = a_{i+2} - X_2 \cdot \eta$

Similarly for the row operation. Z_{20} and Z_{30} are eliminated and replace Z_{10} by - k in each transformation.

For origin shift, at k^{th} stage, we need to consider the last 2 x 2 sub-diagonal matrix. First, find the two roots of this 2 x 2 matrix, $\lambda^{(k)}$, $\lambda^{(k+1)}$ which differ least from the last two diagonal elements respectively. Then compare with the previous two roots $\lambda^{(k-2)}$ and $\lambda^{(k-1)}$, and calculate

$$\left|\frac{\lambda^{(k)} - \lambda^{(k-2)}}{\lambda^{(k)}}\right| \text{ and } \left|\frac{\lambda^{(k+1)} - \lambda^{(k-1)}}{\lambda^{(k+1)}}\right|$$

if both $> \frac{1}{2}$ then set $E^{(k)} = E^{(k+1)} = 0$; if both $< \frac{1}{2}$, then set $E^{(k)} = \lambda^{(k)}$, $E^{(k+1)} = \lambda^{(k+1)}$; otherwise set both $E^{(k)}$ and $E^{(k+1)}$ to be the real part of either $\lambda^{(k)}$ or $\lambda^{(k+1)}$ whichever corresponds to the quantity $< \frac{1}{2}$. Then iterate again until either or both of the last two sub-diagonal elements are near zero, then deflate the matrix accordingly; record the eigen-value or eigen-values; find new origin shifts; iterate again in the same way until all eigen-values are found.

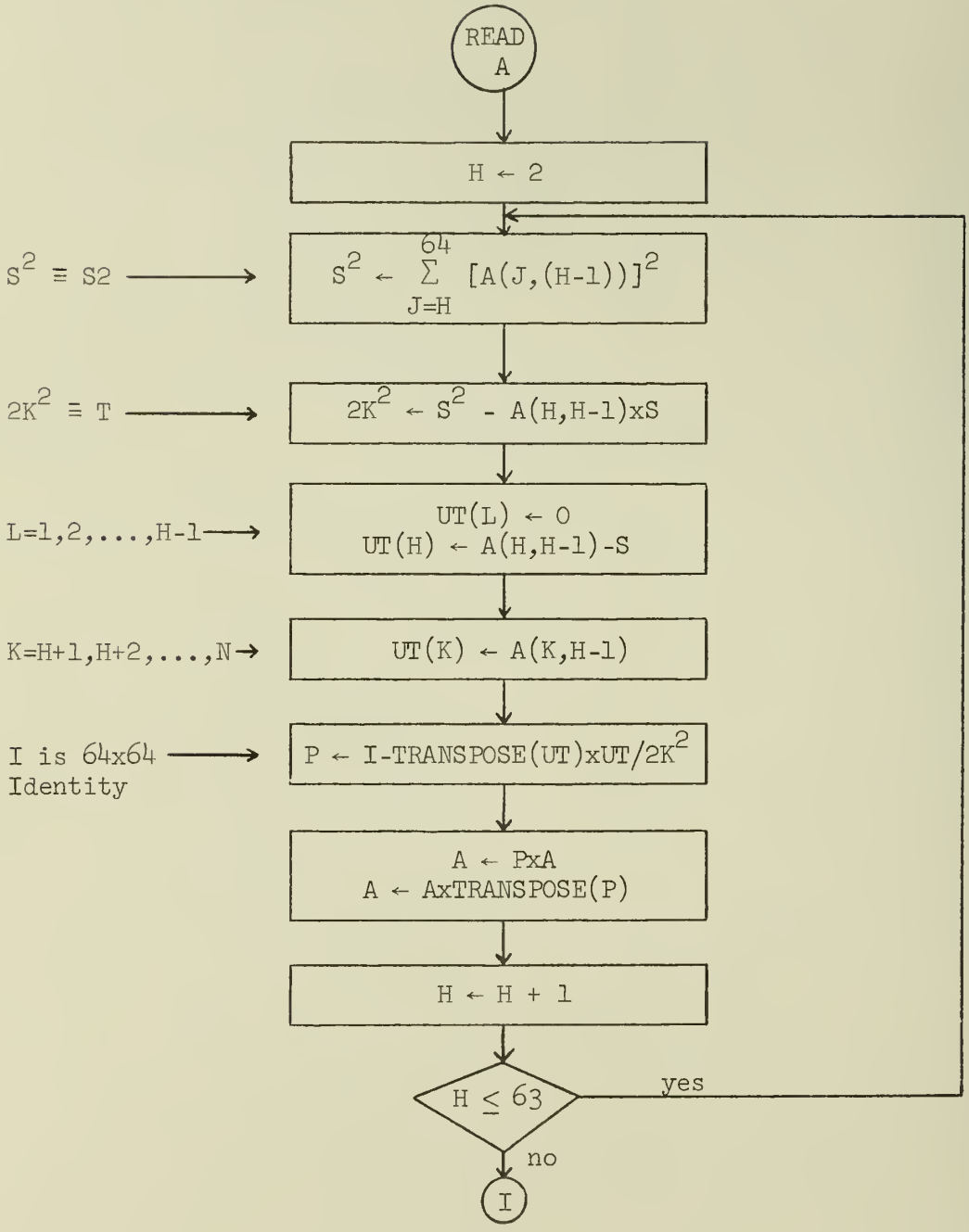
```
"TRANQUIL FOR Q-R ALGORITHM WITH DOUBLE ORIGIN SHIFT

    #BEGIN
        #LABEL    START, LOOP1, LOOP2, LOOP3, ROWOP, CLMOP;
        #ARRAY  #REAL  #FLOAT  #SHORT  #SKWD
                  A[64,64], P[64,64], IM[64,64], UT[64,1];

        #REAL  #FLOAT  #SHORT
                  S2, S, T, E1, E2, TEMP1, X1, X2, L1, L2, EE1, EE2,
                  ESUM, EPDT, R1, R2, R3, K, ALP, P1, P2, AN, RT1, RT2;
        #REAL  #FIXED
                  J, H, K1, L, N, I;
        #SET      HH, LL, KK, II, JJ;
START:    #READ      A;
                  J ← 2;
LOOP1:            HH::[J,J+1,...,64];
                  LL::[1,2,...,J-1];
                  KK::[J+1,J+2,...,64];
                  S2 ← #FOR (H) #SIM (HH) #DO  #SUM(A[H,J-1]↑2);
                  S ← SQRT(S2);
                  T ← S2 - A[J,J-1]* S;
                  #FOR (L) #SIM (LL) #DO UT[L,1] ← 0.;
                  UT[J,1] ← A[J,J-1]- S;
                  #FOR (K1) #SIM (KK) #DO UT[K1,1] ← A[K1,J-1];
                     P ← IM-TRANSPOSE (UT) x UT/T;
                     A ← PxA;
                     A ← A x TRANSPOSE (P);
                     J ← J+1;
                  #IF  J ≤ 63 #THEN  #GOTO  LOOP1;
                  N ← 64;
LOOP2:            II::[1,2,...,N]
                  E1 ← E2 ← 0.;
```

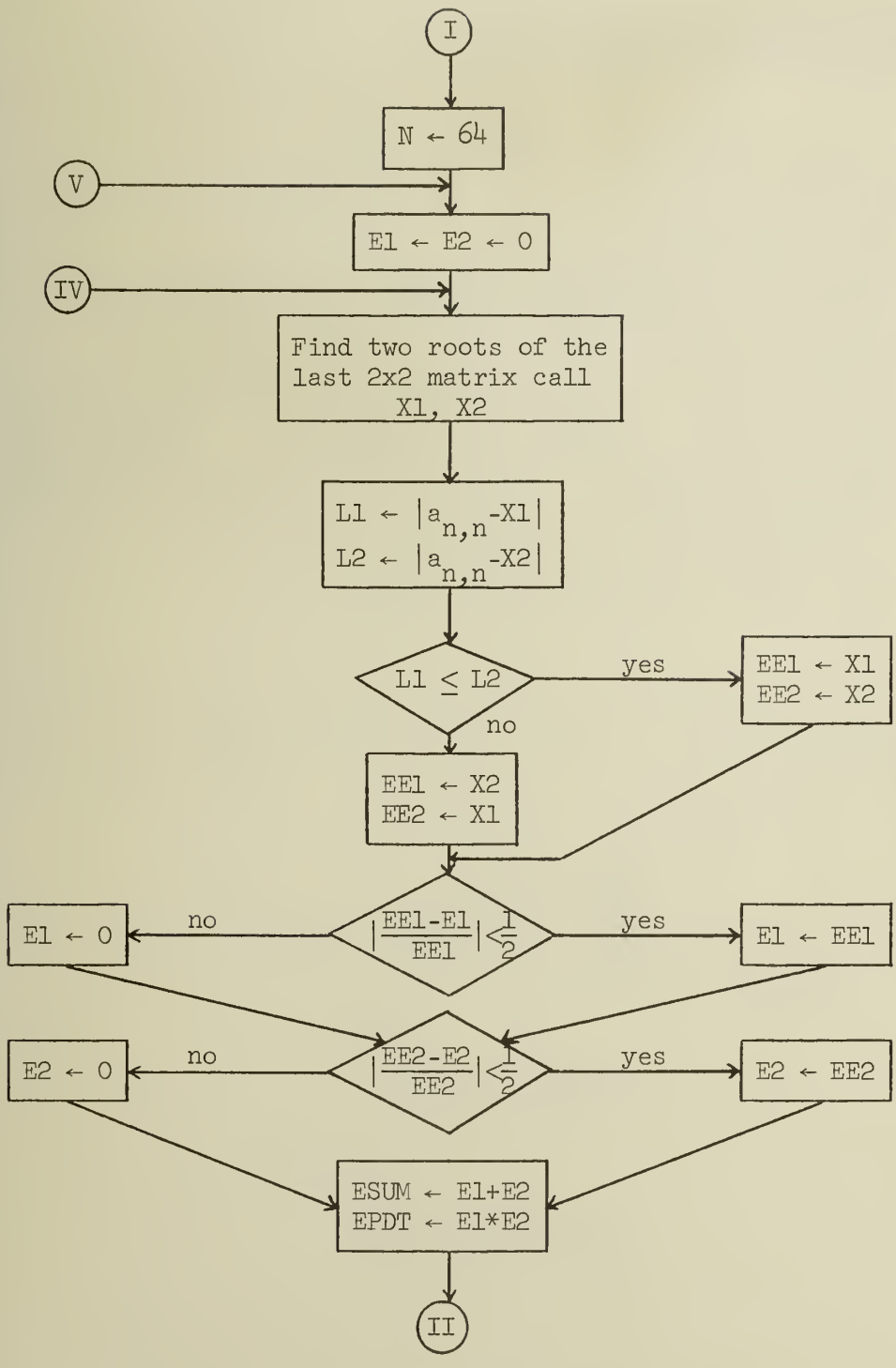
```
LOOP3:      TEMP1 ← SQRT((A[N,N] + A[N-1,N-1])↑ 2-4*(A[N-1,N-1]*
               A[N,N] - A[N,N-1]* A[N-1,N]));
            X1 ← .5* (A[N,N] + A[N-1,N-1] + TEMP1);
            X2 ← .5* (A[N,N] + A[N-1,N-1] - TEMP1);
            L1 ← ABS (A[N,N] - X1);
            L2 ← ABS (A[N,N] - X2);
            #IF L1 ≤ L2  #THEN  #DO  #BEGIN
               EE1 ← X1;
               EE2 ← X2;  #END  #ELSE  #DO  #BEGIN
               EE1 ← X2;
               EE2 ← X1;  #END;
            #IF ABS((EE1-E1)/EE1) < .5  #THEN
               E1 ← EE1  #ELSE
               E1 ← 0.;
            #IF ABS((EE2-E2)/EE2) < .5  #THEN
ROWOP:      #FOR (J)  #SIM(JJ)  #DO  #BEGIN
            #IF I ≤ N - 2  #THEN
               AN ← ALP* (A[I,J] + P1* A[I+1,J] + P2* A[I+2,J])
               #ELSE
               AN ← ALP* (A[I,J] + P1* A[I+1,J]);
               A[I,J] ← A[I,J] - AN;
               A[I+1,J] ← A[I+1,J] - P1* AN;
            #IF I ≤ N-2  #THEN
               A[I+2,J] ← A[I+2,J] - P2* AN;
CLMOP:      #IF I ≤ N-2  #THEN
               AN ← ALP* (A[J,I] + P1* A[J,I+1] + P2* A[J,I+2])
               #ELSE
               AN ← ALP* (A[J,I] + P1* A[J,I+1]);
               A[J,I] ← A[J,I] - AN;
               A[J,I+1] ← A[J,I+1] - P1* AN;
            #IF I ≤ N-2  #THEN
               A[J,I+2] ← A[J,I+2] - P2* AN;
            #END;
```

```
#IF I ≤ N-3 #THEN #DO #BEGIN
   AN ← ALP * P2* A[I+3,I+2];
   A[I+3,I] ← - AN;
   A[I+3,I+1]← - P1* AN;
   A[I+3,I+2] ← A[I+3,I+2] - P2* AN;
   #END;
   E2 ← EE2 #ELSE
   E2 ← 0.;
   ESUM ← E1+E2;
   EPDT ← E1*E2;
#FOR (I) #SEQ (II) #DO
#BEGIN
#IF I=1 #THEN #DO #BEGIN
   R1 ← A[1,1]* A(A[1,1] - ESUM) + A[1,2] * A[2,1] + EPDT;
   R2 ← A[2,1]* (A[2,2] - ESUM + A[1,1]);
   R3 ← A[2,1]* A[3,2];
   A[3,1] ← 0; #END #ELSE #DO #BEGIN
   R1 = A[I,I-1];
   R2 = A[I+1,I-1];
#IF I=N-1 #THEN R3 ← 0 #ELSE R3 ← A[I+2,I-1];
   #END;
#IF R1 ≥ 0 #THEN
   K ← SQRT (R1↑ 2 + R2↑ 2 + R3↑ 2 ) #ELSE
   K ← - SQRT (R1↑ 2 + R2↑ 2 + R3 ↑2);
#IF K ≠ 0 #THEN #DO #BEGIN
   ALP ← R1/K + 1.;
   P1 ← R2/(R1+k);
   P2 ← R3/(R1+k); #END #ELSE #DO #BEGIN
   ALP ← 2;
   P1 ← P2 ← 0.; #END;
JJ::[I,I+1,....,N];
#IF ABS(A[N,N-1]) > .000001 #THEN #DO #BEGIN
#IF ABS(A[N-1,N-2]) > .000001 #THEN #GOTO #LOOP3;
   TEMP1 ← SQRT((A[N,N] + A[N-1,N-1])↑2-4* (A[N-1,N-1]*
      A[N,N] - A[N,N-1]* A[N-1,N]));
```

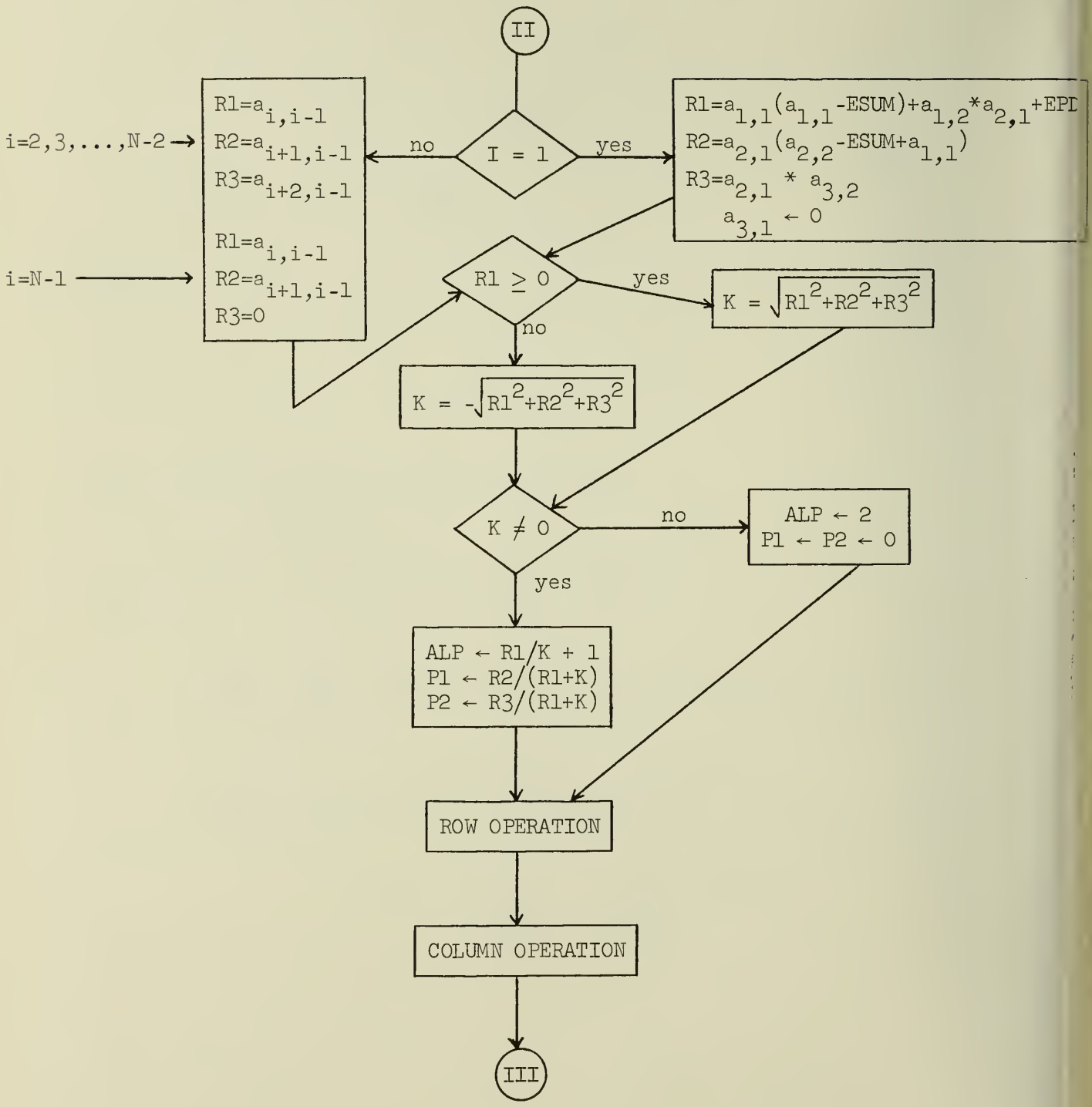
```
            RT1 ← .5* (A[N,N] + A[N-1,N-1] + TEMP1);
            RT2 ← .5* (A[N,N] + A[N-1,N-1] - TEMP1);
         #PRINT  RT1, RT2;
            N ← N-2;  #END  #ELSE  #DO  #BEGIN
         #IF ABS(A[N-1,N-2]) ≤ .000001  #THEN  #DO  #BEGIN
            #PRINT  A[N,N], A[N-1,N-1];
            N ← N-2;  #END  #ELSE  #DO  #BEGIN
            #PRINT  A[N,N];
            N ← N-1;  #END;
         #END;
#END;
         #IF N ≤ 1  #THEN  #GOTO  LOOP2;
         #STOP;
#END
```

TO GET UPPER HESSENBERG FORM

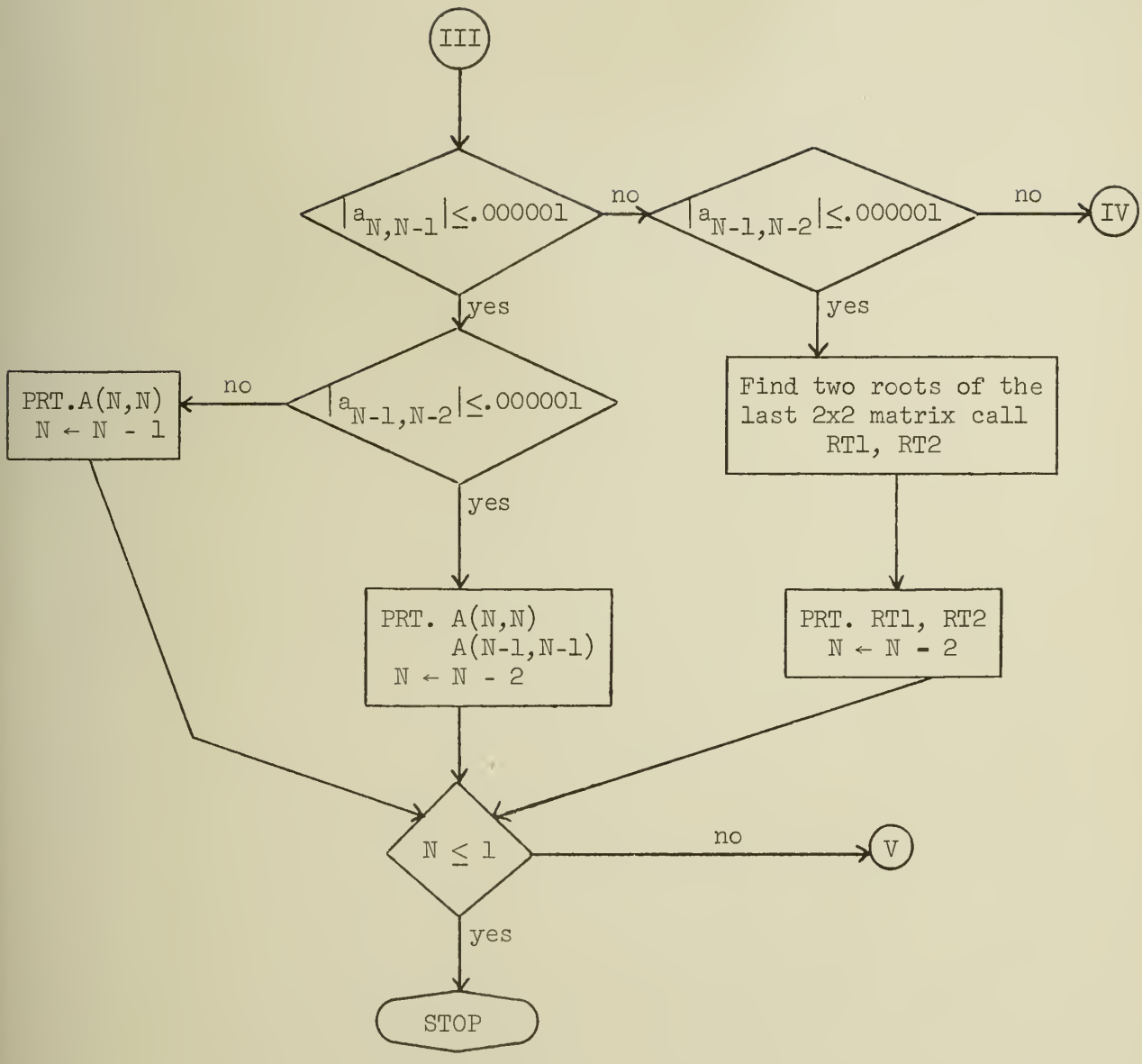



I
N ← 64
V
E1 ← E2 ← 0
IV
Find two roots of the last 2x2 matrix call X1, X2
L1 ← |a n,n -X1|
L2 ← |a n,n -X2|
L1 ≤ L2
yes
EE1 ← X1
EE2 ← X2
no
EE1 ← X2
EE2 ← X1
|EE1-E1 / EE1| ≤ 1/2
no
E1 ← 0
yes
E1 ← EE1
|EE2-E2 / EE2| ≤ 1/2
no
E2 ← 0
yes
E2 ← EE2
ESUM ← E1+E2
EPDT ← E1*E2
II

SET UP AND DO ROW AND COLUMN OPERATIONS



END TEST


III
$|a_{N,N-1}| \leq .000001$
no
$|a_{N-1,N-2}| \leq .000001$
no
IV
yes
yes
$|a_{N-1,N-2}| \leq .000001$
no
PRT.A(N,N)
N ← N - 1
yes
Find two roots of the last 2x2 matrix call RT1, RT2
PRT. A(N,N)
A(N-1,N-1)
N ← N - 2
PRT. RT1, RT2
N ← N - 2
N ≤ 1
no
V
yes
STOP